Avaliação de diagramas no Mooshak 2.0

Helder Patrick De Pina Correia
Mestrado Integrado em Engenharia de Redes em Sistemas Informáticos
Departamento de Ciência de Computadores
2017

Orientador
José Paulo de Vilhena Geraldes Leal, Professor Auxiliar,
Faculdade de Ciências da Universidade do Porto

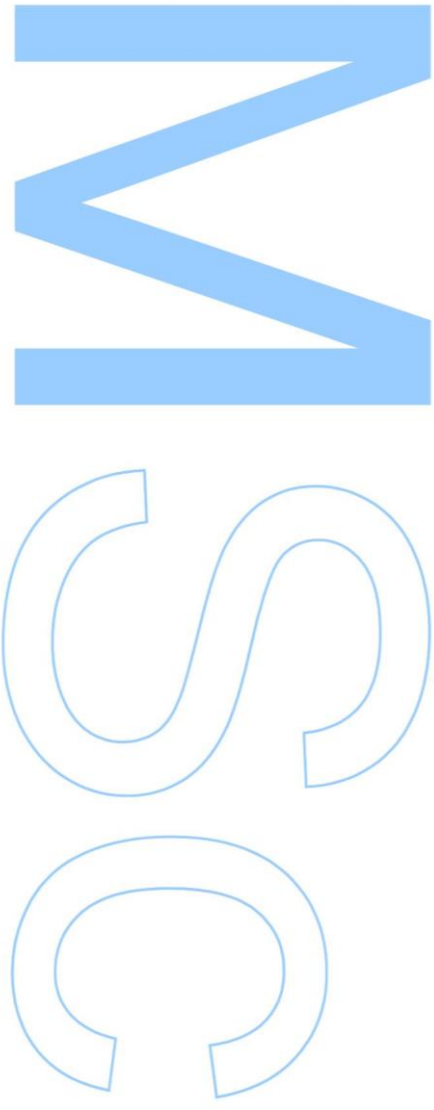
U.PORTO

FACULDADE DE CIÊNCIAS
UNIVERSIDADE DO PORTO

Todas as correções determinadas pelo júri, e só essas, foram efetuadas.

O Presidente do Júri,

Porto, _____/_____/_____



Para os meus pais e para a minha irmã

Agradecimentos

O embarcar numa tarefa desta magnitude corresponde a um esforço que muito dificilmente se pode aguentar sozinho. Apesar de grande parte do esforço ser solitário é reconfortante saber que existem pessoas que estão sempre dispostas a ajudar e apoiar. Esta tese é o resultado de todas essas ajudas, que conduziram-me até aqui.

Ao meu orientador, Prof. José Paulo Leal, pela disponibilidade demonstrada e pelo incentivo constante. As suas sugestões e críticas contribuíram para a melhoria deste trabalho, bem assim como a amizade criada que permitiu que me mantivesse sempre empenhado.

Aos meus colegas, do curso, pelo ambiente de trabalho e companheirismo demonstrado durante o curso. Uma lembrança especial para José Carlos Paiva , com quem trabalhei este projeto e que foi sempre incansável no apoio prestado.

Por fim, nada disto seria possível sem a ajuda da minha família. Aos meus pais, avós e a minha irmã Jussara Correia, pelo apoio e pela compreensão nestes anos de ausências e menor disponibilidade.

Resumo

O Mooshak é um sistema web para avaliação em ciência da computação. Foi originalmente desenvolvido para gerir concursos de programação, mas evoluiu para ser usado também como uma ferramenta pedagógica, capitalizando nos seus recursos de avaliação de programas.

A versão atual do Mooshak suporta outras formas de avaliação utilizadas na ciência de computadores, como a avaliação de diagramas. Esta forma de avaliação é suportada por um conjunto de novas ferramentas, incluindo um editor de diagramas, um comparador de grafos e um ambiente de integração de atividades pedagógicas. A primeira tentativa de integrar essas ferramentas para permitir a avaliação de diagramas revelou uma série de deficiências, como a falta de suporte para múltiplas linguagens diagramáticas, *feedback* ineficaz e problemas de usabilidade.

A finalidade do trabalho descrito nesta dissertação é colmatar estas deficiências. A abordagem seguida passou pela criação de uma linguagem de definição de linguagem diagramáticas (DL^2), pela a introdução de um novo componente para avaliação diagramas, pela gestão das mensagem de *feedback* (Kora) e no redesenho e reimplementação do editor de diagramas Eshu. Para garantir a cobertura dum leque variado de diagramas, foi definido como objetivo o suporte do UML (*Unified Modeling Language*). Além de apresentar o desenho e detalhar a implementação de cada uma das soluções propostas, é ainda descrita a validação do novo ambiente de aprendizagem.

Abstract

Mooshak is a web system with support for assessment in computer science. It was originally developed for programming contest management, but evolved to be used also as a pedagogical tool, capitalizing on its programming assessment features.

The current version of Mooshak supports other forms of assessment used in computer science, such as diagram assessment. This form of assessment is supported by a set of new features, including a diagram editor, a graph comparator, and an environment for integration of pedagogical activities. The first attempt to integrate these features to support diagram assessment revealed a number of shortcomings, such as the lack of support for multiple diagrammatic languages, ineffective feedback, and usability issues.

The purpose of the work described in this dissertation is to fill these deficiencies. The approach followed was the creation of a diagrammatic language definition language (DL^2), the introduction of a new component for evaluation of diagrams, management of feedback (Kora) and the redesign and reimplementation of the diagram editor (Eshu). To guarantee the coverage of a large range of diagrams, UML (Unified Modeling Language) support was defined as the main goal. In addition to presenting the design and detailing the implementation of each of the proposed solutions, the validation of the new learning environment is also described.

Conteúdo

Resumo **5**

Abstract **6**

Lista de Tabelas **13**

Lista de Figuras **16**

1 Introdução **17**

1.1 Motivação 20

1.2 Objetivos 21

1.3 Abordagem 22

1.4 Estrutura da Dissertação 24

2 Background **25**

2.1 A Linguagem UML 25

2.1.1 Objetivos do UML 26

2.1.2 História do UML 26

2.2 Modelação em UML 28

2.2.1 Tipos de elementos básicos 29

2.2.2 Tipos de relacionamentos 30

2.2.3 Tipos de diagramas 31
2.3 Ferramentas UML 33
2.3.1 Ferramentas UML comerciaos 35
2.3.2 Ferramentas UML livres 36
2.3.3 Ferramentas UML web 37
2.4 Sistema de crítica 38

3 Trabalho Prévio **41**
3.1 Eshu - Editor de diagrama 41
3.1.1 Extensibilidade 43
3.1.2 Validação 44
3.2 Enki- Ambiente *web* para aprendizagem de linguagens computacionais . 46
3.3 GraphEval - Avaliador de Diagramas 47

4 Kora **48**
4.1 Design do Kora 49
4.2 Avaliação de diagramas no sistema Kora 50
4.2.1 Avaliação sintática 50
4.2.2 Avaliação semântica 51
4.2.3 Avaliação do diagrama 51
4.3 Geração de feedback 52
4.3.1 Arquitetura e funcionamento do FeedbackManager 53
4.3.2 Características do gerador de feedback 54
4.3.3 Exemplo de feedback gerado pelo KoraManager 54
4.4 Resumo 56

5 Editor de diagrama - Eshu **58**

5.1 Design e Implementação 58
5.1.1 Pacote Graph 59
5.1.1.1 Classe Graph 59
5.1.1.2 Classe Vertice 60
5.1.1.3 Classe Edge 60
5.1.1.4 Classe Quadtree 61
5.1.1.5 Classe RectangleSelection 62
5.1.1.6 Classe TextBox 62
5.1.2 Pacote Eshu 62
5.1.2.1 Classe Eshu 62
5.1.2.2 Classe Events 63
5.1.2.3 Classes Layout e Configuration 63
5.1.2.4 Classe FormatPanel 64
5.1.3 Pacote Command 64
5.2 Interface 65
5.2.1 Janela de propriedades 66
5.3 Extensibilidade 68
5.4 Resumo 69

6 Linguagem de configuração - DL^2 71

6.1 Principais elementos do DL^2 71
6.1.1 Elemento DL2 71
6.1.2 Elemento Diagram 72
6.1.3 Elemento Style 74
6.2 Resumo 74

7 Validação **75**

7.1 Expressividade da linguagem de configuração 75

7.2 Métricas de avaliações 77

7.3 Usabilidade e Satisfação 78

8 Conclusão **80**

8.1 Trabalho Desenvolvido 80

8.2 Trabalho Futuro 81

A Estudo comparativo sobre os editores de diagrama **83**

A.1 Suporte de diagramas 83

A.2 Características de alguns dos editores de diagrama 83

B API da linguagem de configurações – DL^2 **86**

B.1 DL^2 86

B.2 Style 87

B.3 Diagram 88

B.3.1 Nodetype 89

B.3.2 Edgetype 91

B.3.3 SyntaxeValidation 92

B.4 Tipos Auxiliares 93

B.4.1 Label 93

B.4.2 Container 94

B.4.3 Property 95

B.4.4 NodeStyle 96

B.4.5 EdgeStyle 97

B.4.6 Stereotype 98

B.4.7 Connect 98
B.4.8 Anchor 99

C Exemplo de definição de tipo de nó e aresta no DL^2 100

C.1 Exemplo de configuração do nodeType Classe 100
C.2 Exemplo configuração de edgeType – Extend 103

D API do editor de grafo Eshu 105

D.1 Pacote Graph 105
D.1.1 Classe Vertice 105
D.1.2 Classe ComplexVertice 110
D.1.3 Classe Container 113
D.1.4 Classe TextBox 115
D.1.5 Classe Graph 117
D.1.6 Classe Edge 124
D.2 Pacote Eshu 128
D.2.1 Classe Eshu 128
D.2.2 Classe Configuration 133
D.2.3 Classe NodeType 137
D.2.4 Classe EdgeType 139

E Questionário 141

Referências 156

Lista de Tabelas

2.1 Diagramas do UML 33

2.2 Novas diagramas do UML 2.0 34

2.3 Ferramentas do UML Comerciais 36

2.4 Ferramentas do UML livres 37

2.5 Ferramentas do UML para web 38

A.1 Ferramentas UNL e os tipos de diagramas que se pode modelar 84

A.2 Comparação entre os tipo de ferramentas UML 85

D.1 API da classe Vertice – Atributos 105

D.2 API da classe Vertice – Funções 106

D.3 API da classe ComplexVertice 111

D.4 API da classe Container 113

D.5 API da classe TextBox 115

D.6 API da classe Graph 118

D.7 API da classe Edge – Atributos 124

D.8 API da classe Edge – Funções 125

D.9 API da classe Eshu – Atributos 128

D.10 API da classe Eshu – funções 129

D.11 API da classe Events 131

D.12 API da classe Configuration 133
D.13 API da classe NodeType 137
D.14 API da classe Edgetype 139

Lista de Figuras

1.1 Esquema com os interações dos componentes no ambiente de avaliação de diagrama no Enki. 23

1.2 Esquema com os novos componentes e interações destes componentes no ambiente de avaliação de diagrama no Enki. 23

2.1 Visão Histórica do UML, *fonte: UML metodologias e ferramentas case.* 27

2.2 Exemplo dos principais elementos de estrutura do UML 29

2.3 Exemplo dos elementos de comportamento do UML 30

2.4 Exemplo dos elementos de agrupamento do UML 30

2.5 Exemplo dos elementos de anotação e restrição do UML 30

2.6 Exemplo dos principais tipos de relações do UML 31

2.7 Tipos de diagramas do UML 32

3.1 Diagrama de classe do esquema de dados usado pela API. 42

3.2 *Screenshot* do Eshu integrado no Enki 44

3.3 Comunicação do editor de diagrama Eshu com Enki sobre a submissão com o avaliador 45

4.1 Interface do Kora integrado no Enki 48

4.2 Diagrama de pacote do design do Kora 49

4.3 Diagrama de sequência da validação de diagrama no sistema Kora . . . 51

4.4 Diagrama de classe do *FeedbackManager* 53

4.5 Exemplo de diagrama (ER) 55

4.6 Exemplo de feedback visual no diagramas 56

5.1 Diagrama de pacote do Eshu 58

5.2 Diagrama Classe do Eshu 59

5.3 Screenshot da aplicação teste do Eshu 65

5.4 Janela de propriedades 67

5.5 Views da janela de propriedades 68

6.1 Diagrama Classe do DL2 72

7.1 Exemplo de Nós 76

7.2 Avaliação de aceitabilidade - no lado direito os resultados da versão anterior, e no lado direito os resultados da nova versão 78

B.1 Elemento DL2, style e Diagram 86

B.2 Elemento Style, EditorStyle e Toolbar 87

B.3 Elemento Diagram 88

B.4 Elemento Diagram 89

B.5 Elemento Edgetype 91

B.6 Elemento SyntaxeValidation 92

B.7 Elemento Label 93

B.8 Elemento Container 95

B.9 Elemento Property 96

B.10 Elemento NodeStyle 96

B.11 Elemento EdgeStyle 97

B.12 Elemento Stereotype 98

B.13 Elemento Connect 98

B.14 Elemento Anchor 99

Capítulo 1

Introdução

Practice doesn't make perfect. Only perfect practice makes perfect.
Vince Lombardi

A avaliação automática é essencial para que o ensino seja efetivo. Tanto na avaliação formativa quanto na avaliação sumativa, os alunos precisam que os seus exercícios sejam comparados com soluções padrão, para que eles saibam se estão alcançando o resultado esperado. Qualquer forma de avaliação, mesmo que seja apenas uma nota, já é *feedback* para o aluno. No entanto, o *feedback* deve ser mais do que apenas uma distância da solução correta. Os alunos precisam ser orientados, ver evidências de seus erros e receber sugestões para melhorar seu desempenho [24].

Quando o conjunto de todas as respostas possíveis para um exercício é pequeno, a classificação e *feedback* são bastante fáceis de automatizar. Este é, sem dúvida, o motivo pelo qual as perguntas de múltipla escolha (MCQs – *multiple choice questions*) são tão populares no *eLearning*. Na verdade, competências simples e conhecimento superficial podem ser completamente avaliadas usando MCQs, mas em alguns casos são insuficientes. Por exemplo, é impossível avaliar a proficiência de um aluno numa linguagem usando apenas MCQs. Isto é obviamente verdadeiro em línguas naturais, como o inglês ou o português, e é também o caso das linguagens artificiais usadas em informática. Este fato leva ao desenvolvimento de vários sistemas para avaliar quer as linguagens de programação [11] quer as linguagens diagramáticas [3, 39]. No entanto, o *feedback* nestas últimas ainda é um problema amplamente aberto [20].

Tem havido menos investigação relacionada com a avaliação automática de diagramas do que a avaliação de programas, o que é compreensível, uma vez que os programas são mais relevantes do que os diagramas na ciência da computação. No entanto, os

programas são muito mais difíceis de avaliar do que os diagramas, uma vez que a sua semântica é mais complexa. Ou seja, um programa tem uma semântica operacional que precisa ser verificada com dados de teste, mas os diagramas possuem apenas uma semântica declarativa. O *feedback* a um exercício sobre diagramas pode ser unicamente baseado nas diferenças entre o diagrama do aluno e uma solução. Sendo assim, a relevância da investigação sobre o *feedback* na avaliação do diagrama é dupla: as linguagem de diagrama são estudadas em várias disciplinas de ciência da computação, como a teoria da computação (ex: DFA – *Deterministic finite automaton*), base de dados (ex: EER – *Enhanced entity–relationship*) e modelagem de software (ex: UML – *Unified Modeling Language*), por isso é útil para o ensino dessas matérias; as ferramentas e técnicas desenvolvidas para linguagens diagramáticas podem ser posteriormente estendidas a linguagens mais complexas, como linguagens de programação.

Esta tese baseia-se em projetos anteriores que conduziram ao desenvolvimento dos componentes usados na avaliação de diagramas, nomeadamente um ambiente de aprendizagem de linguagem de computação [33], um editor de diagrama [21] e um comparador de grafo [40]. Os diagramas são modelados por grafos, portanto, é possível comparar dois diagramas calculando as diferenças entre os grafos do modelo. Para grafos grandes, a complexidade computacional é proibitiva, mas usando heurísticas e para grafos do tamanho tipicamente usado nos exercícios de programação, esse método é eficaz. No entanto, a validação do comparador de grafos revelou várias questões relacionadas à geração de *feedback*.

A avaliação realizada pelo comparador de grafos pode ser descrita como semântica. Ou seja, o grafo é uma representação semântica do diagrama e as diferenças entre os dois grafos refletem as diferenças de significado dos dois diagramas. No entanto, em muitos casos, as diferenças resultam do fato de que a tentativa do aluno não é um diagrama válido. Um erro típico é um diagrama que não gera um grafo totalmente conexo, o que não é aceitável na maioria dos linguagens diagramáticas. Outros erros são específicos da linguagem e referem graus inválidos, ou arestas que ligam tipos de nó incorretos. Por exemplo, num diagrama EER(*Enhanced entity–relationship*), um nó de atributo só pode ser ligado por uma única aresta e dois nós de entidade não podem ser ligados diretamente. Assim, o *feedback* será mais eficaz se indicar esse tipo de erros e remeter ao aluno para uma página ou vídeo descrevendo essa parte específica da linguagem. Para permitir esse tipo de *feedback* sintático, o Kora fornece uma linguagem de definição diagramática, que também pode ser usada para relacionar erros detetados com o conteúdo disponível que pode ser fornecido como *feedback*.

Outro problema com o comparador de grafos é que frequentemente reporta demasiadas

diferenças. Uma situação semelhante ocorre com erros sintáticos reportados por um compilador. O *feedback* detalhado em grandes quantidades geralmente é menos útil do que um *feedback* conciso sobre os problemas mais relevantes. Por exemplo, ao avaliar um diagrama de EER(*Enhanced entity–relationship*), uma única linha de *feedback* que reporta n atributos em falta é mais útil do que n linhas dispersas que reportam cada atributo em falta. No entanto, se o aluno persistir no erro, repetir a mesma mensagem não é útil. A divulgação progressiva do *feedback* deve levar em consideração a informação fornecida ao aluno para evitar repetições desnecessárias. Assim, novos comentários sobre os mesmos erros devem focar-se em questões cada vez mais específicas e fornecer progressivamente mais detalhes. Além disso, esse *feedback* incremental deve ser parcimonioso para desencorajar os alunos de o usar como uma espécie de oráculo e evitar pensar por si mesmos.

No ensino de programação antes de ensinar a sintaxe e as propriedades de uma linguagem, muitas vezes procura-se transmitir a ideia da lógica de programação [12]. Para isso, utiliza-se as linguagem diagramáticas como por exemplo fluxogramas, uma linguagem visual, em que os comandos são representados por figuras e setas [5]. Estas linguagem permitem explicar de forma simples os programas e os conceitos sobre um sistema. Contudo, apesar dessas linguagens diagramáticas serem simples é preciso estuda-las e ter conhecimento para poder interpreta-las. No entanto, não há muitas ferramentas de avaliação automática de diagramas com o *feedback* adequado de modo a ajudar na aprendizagem, e a maioria delas foram projetadas para um tipo de diagrama específico [3, 39].

O Kora é um componente que complementa o Enki [33], uma ferramenta de aprendizagem existente para cursos de programação, com a capacidade de modelar e avaliar diagramas de qualquer tipo, como os diagramas *Unified Modeling Language* (UML), *Extended Entity-Relationship* (EER), *Deterministic Finite Automaton* (DFA), entre outros. Ele fornece *feedback* visual e textual aos alunos, identificando os seus erros e orientando-os para uma solução com sugestões sobre os próximos passos.

Durante a elaboração deste trabalho foram publicados e apresentados dois artigos científicos. O primeiro artigo, a qual foi atribuído o prémio de melhor artigo na área de *Computer-Computer Languages* e cujo o titulo é *Enhancing feedback to students in automated diagram assessment* Vila do Conde (Portugal) [7], conferência Slate'17 e um segundo artigo titulado de *Improving Diagram Assessment in Mooshak*, conferência TEA em Barcelona (Espanha) [8].

O Mooshak com Kora está disponível para download na página inicial do projeto.

Também está disponível para teste online, uma instalação do Mooshak configurada com alguns exercícios de diagrama ER (em inglês) [1].

1.1 Motivação

O erro não é fonte de castigo, mas suporte para o crescimento.
Luckesi (2002)

A resolução de exercícios é fundamental para a aprendizagem de uma nova linguagem. Contudo, devido à ignorância natural de quem está a aprender, são cometidos muitos erros durante essa aprendizagem. Esses erros são parte integrante do processo, e se corretamente corrigidos, são uma parte importante da aprendizagem. Por isso, é determinante a qualidade de *feedback* fornecido para ajudar a compreender e corrigir os erros. Quanto mais específicas e claras forem as correções dos erros, melhor será a ajuda e maior a motivação do aluno [38]. No entanto, é difícil para os professores fornecer sempre a quantidade correta de *feedback* a cada aluno.

As ferramentas de avaliação automática podem preencher essa lacuna. Estas ferramentas ajudam os professores a avaliar múltiplas tentativas dos alunos em pouco tempo. No entanto, a maioria das ferramentas de avaliação fornece apenas uma nota ou classificação a uma tentativa de um aluno, o que é insuficiente quando comparado com o *feedback* do professor. Por isso, neste trabalho o foco é desenvolver uma ferramenta que permita editar e avaliar diagramas, identificar erros e pontos de melhoria, e fornecer sugestões para os ultrapassar.

Como representantes das linguagens diagramáticas foram escolhidos o UML (*Unified Modeling Language*) e EER (*Extended Entity–Relationship*), ambas bastante usadas na ciência de computadores. A linguagem EER já era anteriormente suportada e foi mantida, o que facilitará a comparação com versão anterior do ambiente de avaliação de diagramas. A linguagem UML foi escolhida por apresentar uma grande variedade de tipos de diagrama e terá, por isso, maior destaque nesta dissertação.

Os diagramas podem ser encarados como a visualização de um grafo e, portanto, são constituídos por elementos grafos que representam nós e arestas. A maioria dos tipos de nós dos diagramas são simples do ponto de vista da sua visualização e interação. Isto é, são definidos por uma imagem e apresentam uma etiqueta (*label*) para editar o

[1] http://mooshak2.dcc.fc.up.pt/Kora

seu nome. Por exemplo, num diagrama de casos de uso do UML um ator é representado por um boneco estilizado e um caso de uso por uma oval, cada qual com a sua etiqueta. Contudo, existem alguns tipos de nós complexos cuja implementação apresenta um maior desafio. Por exemplo, o nó classe do diagrama de classes é constituído por painéis opcionais, para atributos e operações, e cada um destes tem representações textuais complexas.

As arestas são geralmente visualizadas como linhas ligando as visualizações dos respetivos nós, frequentemente com uma ponta de seta, como é o caso das generalizações e das dependências nos diagramas de classes. No entanto, pode haver tipos de arestas que não são explicitamente representadas como ligações. Por exemplo, no diagrama de caso de uso o nó do tipo sistema é representado como um retângulo que não é ligado a outros elementos. Os casos de uso *contidos* no retângulo são implicitamente ligados ao sistema.

1.2 Objetivos

A finalidade deste projeto é desenvolver um ambiente de aprendizagem de linguagens diagramáticas, com base na resolução de exercícios práticos. Este ambiente deve cumprir os seguintes objetivos genéricos.

- Desenvolver um ambiente de avaliação de diagramas com suporte para múltiplas linguagens diagramáticas.
- Estimular a aprendizagem das linguagens diagramáticas – fornecer aos alunos um mecanismo que lhes ajude a ultrapassar as dificuldades na aprendizagem das linguagens diagramáticas.
- Flexibilidade no tempo e local - a ferramenta deve ser acessível a qualquer altura e em qualquer lugar (levar a escola para fora da escola) através da Internet. Ter um mecanismo sempre disponível a ajudar na aprendizagem das Linguagens diagramáticas.

Desta finalidade derivam os seguintes objetivos específicos:

- Desenvolver um editor de diagramas destinado aos alunos que, através de exercícios práticos, querem aprender mais sobre uma linguagem diagramática.

- Avaliar automaticamente exercícios das linguagens diagramáticas - a ferramenta, além de ajudar os professores no ensino das linguagens diagramáticas, permite-os avaliarem de forma rápida e uniforme (mesmo critério) os conhecimentos dos seus alunos sobre essas linguagens.

- Desenvolver um sistema *feedback* incremental e adaptativo ao nível do conhecimento ou objetivo do aluno na edição de diagramas - a ferramenta deve oferecer múltiplos níveis de *feedback* na resolução dos exercícios práticos sobre as linguagens diagramáticas e adaptar o *feedback* ao longo da evolução da aprendizagem do aluno, de modo a complementar a parte teórica dessas linguagens.

1.3 Abordagem

Com os objetivos apresentados acima, pretende-se ter como resultado final deste trabalho uma ferramenta constituída por um editor de diagramas, um gestor de *feedback* e um avaliador de diagramas. A ferramenta será integrada no Mooshak 2.0 [33] de modo a promover a aprendizagem de linguagens diagramáticas com base na resolução de exercícios práticos – ajudar os alunos a desenvolver a capacidade em projetar modelos de sistemas válidos de forma autónoma e correta. Para isso, abordamos os seguintes recursos para a realização desta ferramenta:

- **Eshu 2.0**: a versão 2.0 do Mooshak possui um editor de diagramas e um ambiente de avaliação de grafos. Estes dois componentes já foram integrados para criar um ambiente de aprendizagem de diagramas ERR(*extended entity–relationship*). Pretende-se desenvolver uma nova versão do editor de diagrama (Eshu) extensível a novas linguagens diagramáticas.

- **Kora**: pretende-se melhorar a integração de *feedback* no Eshu desenvolvendo, para isso, um gerador de *feedback* flexível e adaptável e interligar com o editor de diagrama. Este sistema, o qual designamos de Kora, irá atuar sobre o editor de diagramas fornecendo correções e sugestões após submissão de diagramas, com o objetivo de ajudar o aluno a desenvolver diagramas válidos. Também ira atuar como um sistema de *parser*, e interligação entre Eshu e Avaliador de diagramas.

- **Linguagem de configuração**: Pretende-se desenvolver uma linguagem de configuração (DL^2) de modo a poder estender o editor e o gerador de *feedback* Kora a novos tipos de linguagens diagramáticas.

- **Avaliador diagrama**: o avaliador de exercício baseado em diagramas [40] recebe um diagrama com uma tentativa de resolução de um problema e o compara com um diagrama solução e retornas as diferenças e uma classificação.

- **Ferramenta final**: Integrar estes quatros componentes no Mooshak 2.0 (Enki) de modo a criar um ambiente de aprendizagem para linguagem diagramáticas.

Na figura 1.1 é apresentado um diagrama de componente com os componentes que fazem parte do ambiente de avaliação de diagrama no Enki antes da realização deste trabalho. Este ambiente é formado por Eshu – editor de diagrama e GraphEval – avaliador de grafos.

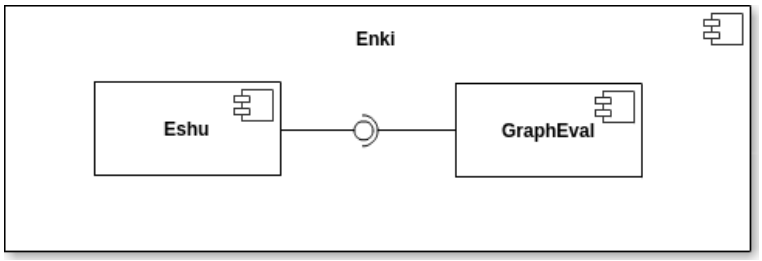


Figura 1.1: Esquema com os interações dos componentes no ambiente de avaliação de diagrama no Enki.

A figura 1.2 apresenta um diagrama de componente com os componentes que fazem parte da nova arquitetura proposta para o ambiente de avaliação de diagrama no Enki. Além dos componentes já existentes, Eshu e GraphEval foram introduzidos os componentes Kora e DL^2.

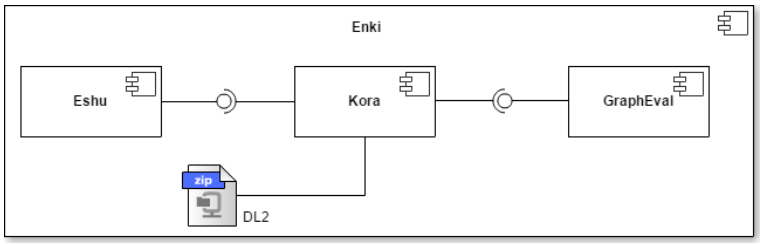


Figura 1.2: Esquema com os novos componentes e interações destes componentes no ambiente de avaliação de diagrama no Enki.

1.4 Estrutura da Dissertação

A presente dissertação é composta por 8 capítulos. Este capítulo descreve os objetivos propostos, a motivação da realização desse projeto e apresenta-se a abordagem ao trabalho.

O capítulo 2 apresenta uma descrição da linguagem UML, versão 2.0. São enunciadas as características mais importantes da linguagem, os elementos, relações, tipos de diagramas, com principal atenção para os diagramas mais relevantes para este trabalho, nomeadamente os diagramas classe e os diagramas caso de uso por serem os mais usados. São apresentados um estudo sobre as ferramentas UML e um resumo sobre os sistemas de crítica.

O capítulo 3 descreve os trabalhos já realizados e que serão utilizados como base para a realização deste projeto. Descreve ainda a extensibilidade e validação do editor de diagrama (o Eshu1.0 [21]), o avaliador de exercícios baseados em diagramas [40] e Enki [34], um ambiente web integrado para aprendizagem de linguagens computacionais.

O capítulo 4 descreve o design, implementação e o tipo de avaliações realizadas pelo componente Kora, (sintática e semântica). Além disso é apresentado a arquitetura do gerador de *feedback*, as características do *feedback* gerado e alguns exemplo de *feedback* gerados.

O capítulo 5 descreve o redesenho e a reimplementação do editor de diagramas Eshu, abordando também a parte do interface e extensibilidade.

O capítulo 6 apresenta os principais elementos e sub-elementos da linguagem de configuração DL^2 que configura o editor de diagrama Eshu e define as regras sintaxe para a validação sintática do diagrama no Kora.

Capitulo 7 apresenta a parte da validação do trabalho realizado. São apresentados os resultados decorrentes das experiências realizadas com alunos e problemas reais.

Por último, são apresentados cinco apêndices: o primeiro apresenta um estudo comparativo das ferramentas UML, o segundo apêndice apresenta os elementos da linguagem de configuração DL^2 em pormenor, o terceiro apêndice apresenta a descrição da API do editor de diagrama Eshu, o quarto um exemplo de definição do nó e aresta a partir da linguagem configuração DL^2 e por fim um apêndice com o questionário utilizado na validar a parte da usabilidade e satisfação.

Capítulo 2

Background

2.1 A Linguagem UML

O UML é o acrónimo de *Unified Modeling Language*, que pode ser traduzido por Linguagem de Modelagem Unificada e é uma linguagem normalizada para a construção de modelos de sistemas *software*. O UML é uma linguagem gráfica que utiliza uma notação padrão para especificar, visualizar, construir e documentar sistemas de informação orientada por objetos [37].

O UML é uma linguagem e não uma metodologia de desenvolvimento. Teve como ênfase a definição de uma linguagem de modelação padrão e, por conseguinte, o UML é independente da linguagem de programação, das ferramentas e processos de desenvolvimento.

Comparativamente a outros métodos existentes, o UML alarga o âmbito em relação à aplicação alvo (*software*, aplicações *web*), pois permite modelar uma grande variedade de sistemas e está concebida para poder ser atualizada de modo a satisfazer qualquer requisito de modelação.

Pela abrangência e simplicidade dos conceitos utilizados, o UML facilita o desenvolvimento de qualquer sistema baseado em *software*, uma vez que, pelo facto de utilizar um conjunto de símbolos padrão, funciona como um meio de comunicação entre os diversos elementos envolvidos no processo, nomeadamente os utilizadores, gestores de equipas e equipa de desenvolvimento. A linguagem pode ser usada para documentar o sistema ao longo do ciclo de desenvolvimento, inicialmente na fase de planeamento, organização e prolonga até à fase de manutenção.

O UML é promovido como padrão pelo *Object Management Group* (OMG), com contribuição de diversas empresas da indústria de *software*. Atualmente, é adotado pelas empresas e instituições de todo mundo, existindo dezenas de ferramentas comerciais e académicas para modelação de *software*.

2.1.1 Objetivos do UML

O UML é uma linguagem gráfica que tem como objetivo a visualização, especificação, construção e documentação de artefactos de sistemas baseados *software*.

- **Visualização:** Por ser uma linguagem gráfica, o UML facilita a visualização do conjunto, pois omite detalhes e destaca as relações entre os componentes (tira partido da imagem como elemento de comunicação). Apresenta um certo rigor na representação gráfica, isto é, os diagramas não são esboço, tem uma semântica e são convertíveis em texto (documento XMI).
- **Especialização:** É um modelo, isto é, uma abstração sobre a realidade, evita detalhes, permite operações simples sobre um todo complexo. Modelar simplifica a realidade, ajuda a compreender o funcionamento de um sistema.
- **Construção:** Pode-se utilizar o UML para *forward engineering* (ex: geração de código automático), *reverse engineering* (ex: reconstrução de modelos a partir de código) e partilha de modelos entre ferramentas via XMI.
- **Documentação:** Permite a partilha de informação e a comunicação entre os intervenientes do projeto (clientes, gestores de projetos, analistas, programadores e testadores).

2.1.2 História do UML

Na figura 2.1 é apresentado um enquadramento histórico da linguagem UML no contexto das linguagens de modelação de *software* orientado a objetos.

No início da década de 1990, dado um aumento de métodos e notação à modelação segundo a abordagem orientada por objetos (fase da fragmentação),surgiram diver-

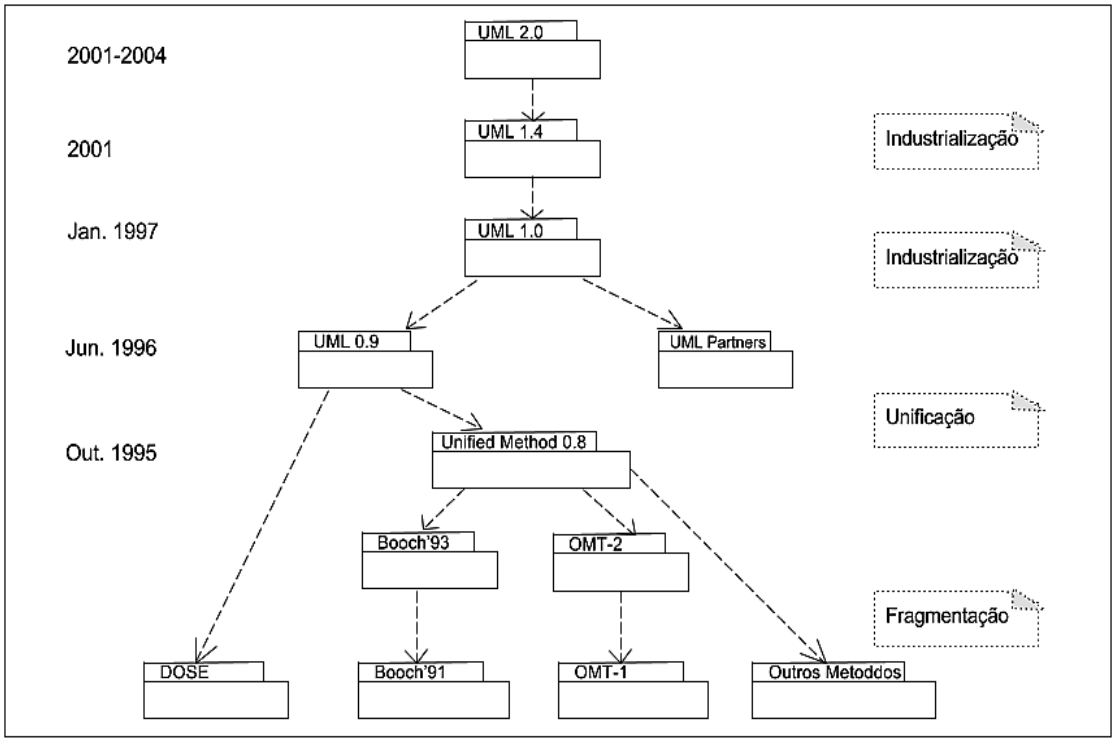


Figura 2.1: Visão Histórica do UML, *fonte: UML metodologias e ferramentas case.*

sidades das nomenclaturas utilizadas nos modelos semânticos, notação sintática e diagramas que tornaram difícil a comunicação e reduzia a utilização prática desta área de conhecimento. Percebeu-se a necessidade da existência de uma linguagem padrão, que fosse aceite e utilizada quer no ambiente industrial como no ambiente académico e de investigação.

Atentos a este problema, os três mais relevantes autores neste domínio – Booch, Jacobson e Rumbaugh – começaram a trabalhar em conjunto para apresentar uma proposta unificadora dos seus trabalhos individuais. Trabalhos estes: métodos de Booch, OMT e OOSE. A primeira fase ficou conhecida como fase da unificação.

Em 1997, este trabalho e com vários outros contributos veio a dar origem à UML, que foi adotada pelo OMG (*Object Management Group*) como uma linguagem de modelação padronizada e de livre utilização. Nos anos seguintes e com sucessivos refinamentos e melhorias na especificação, novas versões (1.1 - 1.5) foram surgindo. A segunda fase ficou conhecida como fase normalização ou estandardização.

A seguir a essa fase, assiste-se à divulgação e adoção generalizada do UML como a linguagem de modelação de *software* segundo a abordagem orientado por objetos.

Começaram a aparecer vários estudos, publicações, artigos dedicados ao UML, para além de ferramentas CASE que suportam a UML. Esta fase foi a fase de industrialização.

Em 2004, surgiu uma nova especificação para UML, conhecida como UML 2.0. A grande inovação foi a implementação da *Model Driven Architecture (MDA)*, uma nova maneira de escrever especificações e de desenvolver aplicações, baseadas em modelos independentes da plataforma (PIM).

A especificação de UML 2.0 é composta por quatro documentos [32]. São eles:

- **Infra-estrutura de UML:** O conjunto de diagramas de UML constitui uma linguagem definida a partir de outra linguagem que define os elementos construtivos fundamentais. Esta linguagem que suporta a definição dos diagramas é apresentada no documento infra-estrutura de UML.
- **Superstrutura de UML:** Documento que complementa o documento de infra-estrutura e que define os elementos e disponibiliza funcionalidades para a construção dos modelos.
- **Linguagem para Restrições de Objetos (OCL):** Documento que apresenta a linguagem usada para descrever expressões em modelos UML, com pré-condições, pós-condições e invariantes.
- **Intercâmbio de diagramas de UML:** Apresenta uma extensão do metamodelo voltado a informações gráficas. A extensão permite a geração de uma descrição no estilo XMI orientada a aspetos gráficos que, em conjunto com o XMI original, permite produzir representações portáveis de especificações UML.

O UML 2.0 veio permitir a modelagem de sistemas em tempo real, através da introdução de novos diagramas de Temporização e Sequência. Foi também introduzido no UML 2.0 os *Profiles* (pacotes que contêm elementos de modelos que foram criados relativamente a uma determinada área), e também permitiu aumentar a capacidade de interligação entre os diagramas.

2.2 Modelação em UML

Conceptualmente o UML é definido por três elementos fundamentais: os elementos básicos, relações e diagramas [37]. Os elementos básicos (também designados como

coisas-"*things*") são as identidades com base nas quais se definem os modelos, as relações definem a forma como elas se relacionam entre si (as regras de relacionamento), enquanto os diagramas definem regras de agrupamentos de elementos.

2.2.1 Tipos de elementos básicos

Os elementos básicos encontram-se organizados consoante as suas funcionalidades ou responsabilidades. Assim sendo, encontram-se subdivididos em elementos estruturais, de comportamento e de anotação.

Entre os elementos estruturais encontram-se as entidades da linguagem, as que constituem um elemento chave dos diagramas (correspondem aos substantivos numa frase). Como exemplos dos elementos estruturais temos as classes, as interfaces, os casos de usos, os objetos, os nós, entre outros.

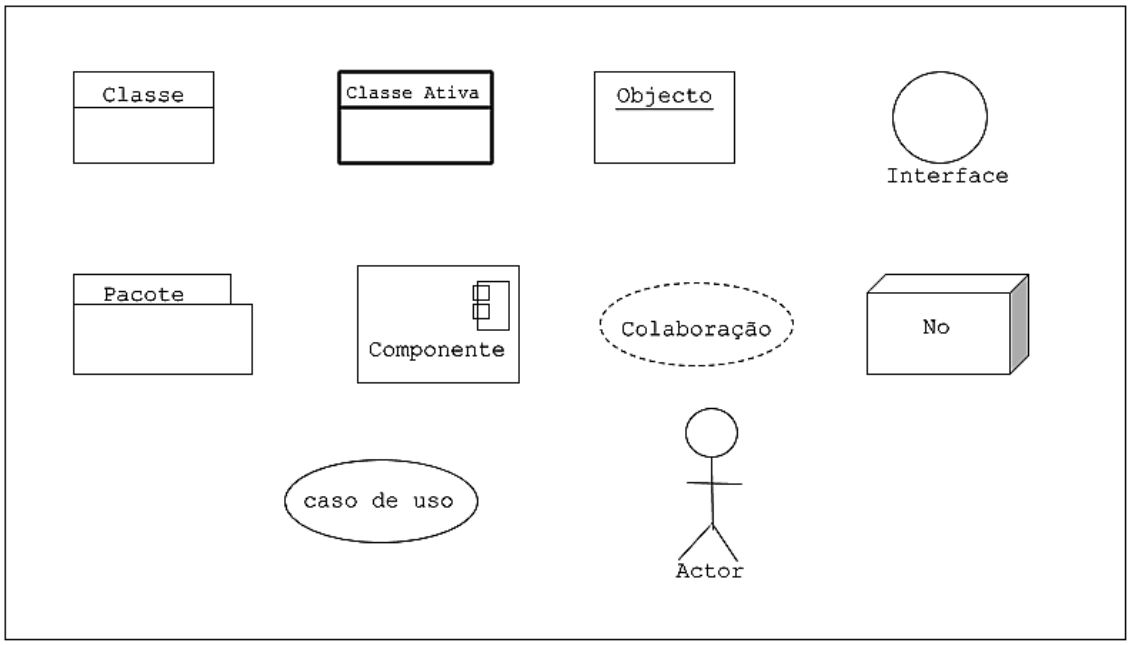


Figura 2.2: Exemplo dos principais elementos de estrutura do UML

No que respeita aos elementos comportamentais, representam a componente dinâmica dos modelos, ações, como por exemplo, uma sequência de mensagens, as iterações, as máquinas de estados que definem a sequência de estados pelos quais as identidades passam.

Os elementos de agrupamentos são mecanismos de organização dos modelos de UML e representam as associações lógicas ou físicas de entidades. Um exemplo destes

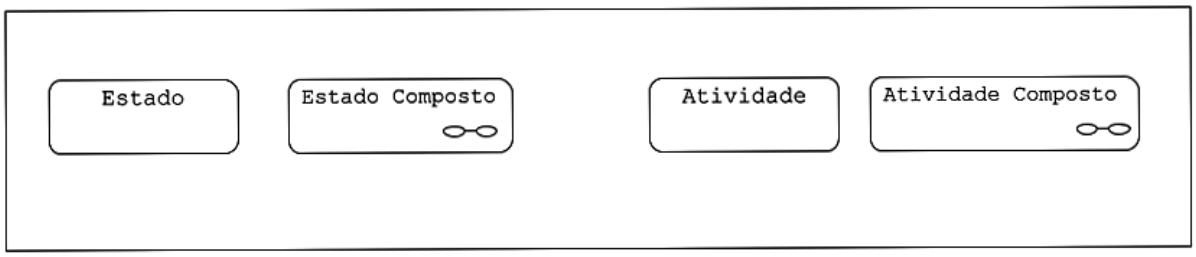


Figura 2.3: Exemplo dos elementos de comportamento do UML

elementos são os pacotes.

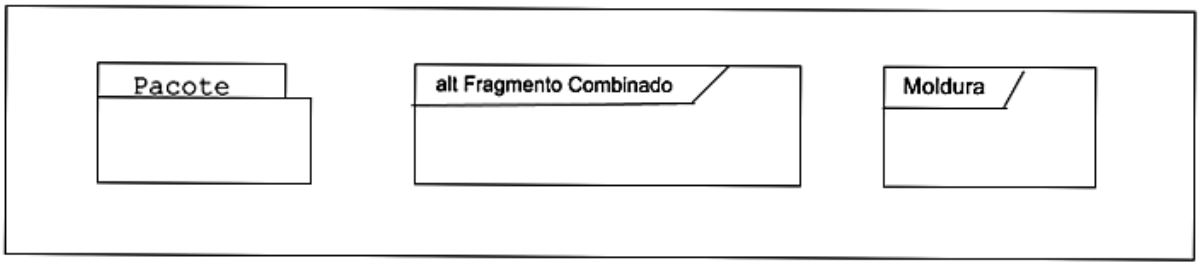


Figura 2.4: Exemplo dos elementos de agrupamento do UML

Os elementos de anotação correspondem à parte da linguagem UML que recorre a notações, em geral, não formais, para melhorar ou especificar detalhes do modelo, fornecem comentários e descrições. Um exemplo é o componente visual nota (*note*) que nela escrevem as notas que ajudam a compreender os modelos.

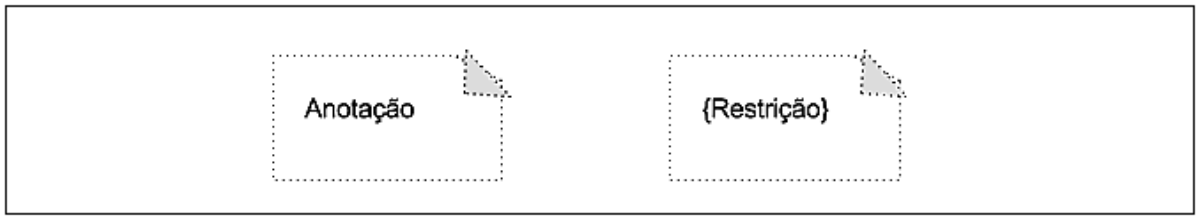


Figura 2.5: Exemplo dos elementos de anotação e restrição do UML

2.2.2 Tipos de relacionamentos

Os relacionamentos são classificados em quatros tipos básicos: **dependência, associação, generalização e implementação**.

- **Dependência** – detalha relacionamentos semânticos entre elementos básicos

indicando que alterações numa extremidade da relação afetam a outra extremidade, isto é, descreve situações em que a mudança num componente afeta outro. Representada graficamente por uma linha tracejada e uma seta a indicar o sentido da dependência e pode ter etiqueta.

- **Associação** – descreve um conjunto de ligações entre objetos, podendo ser decorada com informação sobre a cardinalidade das ocorrências dos objetos. Como casos particulares de associações temos a agregação e a composição de objetos. Representada graficamente por uma linha sólida, geralmente com adornos, que podem ser papéis (etiquetas) e multiplicidade.

- **Generalização** – define uma relação estrutural de hierarquia entre as identidades, ou seja, uma relação que liga um elemento especializado a um elemento generalizado. Basicamente, descreve uma relação de herança no mundo dos objetos. Representada graficamente por uma linha sólida com uma seta aberta.

- **Implementação** – é uma relação semântica entre um conceito e a sua concretização. Um elemento descreve algumas responsabilidades que não são implementadas e a outra as que são implementas. Como exemplo, na linguagem Java temos a relação entre uma interface e a classe que a realiza implementando os métodos aí definidos.

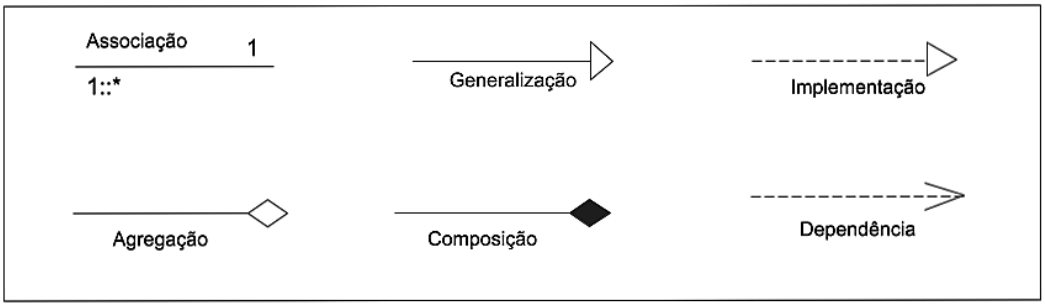


Figura 2.6: Exemplo dos principais tipos de relações do UML

2.2.3 Tipos de diagramas

Um diagrama UML é uma representação gráfica, tipificada e organizada de um conjunto de elementos básicos e as suas relações que constituem uma vista do sistema. Para cada tipo de diagrama é usado um conjunto dos elementos básicos descritos e ligados por diferentes tipos de relações que sejam aplicadas.

Nenhum diagrama é suficiente para representar um sistema não trivial. Cada diagrama fornece uma vista diferente do modelo. Assim um único diagrama não é suficiente para representar todo o modelo do sistema. Deste modo o UML define vários tipos de diagramas que auxiliam na compreensão e modelação dos aspetos mais importantes de um sistema. Normalmente, são divididos em duas grandes categorias [32]: **estruturais e comportamentais**.

- **Diagramas estruturais** — representam a parte estática do sistema, onde detalha a arquitetura do sistema, tanto lógica como física.
- **Diagramas comportamentais** — representam a parte dinâmica do sistema, isto é, as características dinâmicas de um sistema.

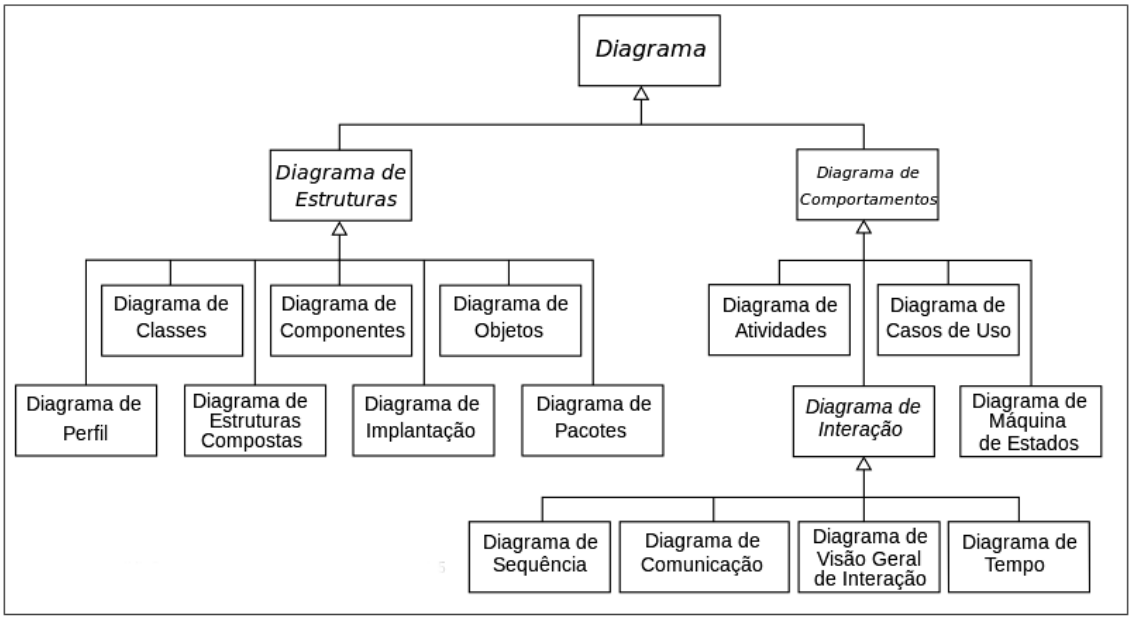


Figura 2.7: Tipos de diagramas do UML

Figura adaptada de UML *Superstructure Specification* 1.4.1, Figure A.5

Nem todos estes diagramas oferecem o mesmo grau de utilidade na descrição de um sistema complexo. Os mais importantes são:

- Diagramas de classes (*class diagrams*);
- Diagramas de caso de uso (*use case diagrams*);
- Diagramas de estado (*statechart diagrams*);

- Diagramas de sequência (*sequence diagrams*);
- Diagramas de atividade (*activity diagrams*);
- Diagramas de objetos (*object diagrams*).

Apesar destes serem os mais relevantes, existem mais diagramas previstos na notação. Nos treze diagramas existentes no UML 2.0, nove foram definidos na versão 1.0 e os restantes introduzidos na nova versão. A tabela 2.1 apresenta os diagramas previstos da notação original (versão 1) e a tabela 2.2 apresenta os restantes diagramas introduzidos ou alterados na versão 2.0.

Diagrama De	Descrição
Classes	Ilustra a natureza estática do sistema, isto é, através da qual descrevemos a estrutura de informação (classes e relações) que é utilizada no sistema.
Casos de uso	Modelam os aspetos dinâmicos de um sistema, identifica os casos de utilização, assim como os atores e funcionalidades por estes invocados.
Estado	É utilizado para modelar o comportamento dos objetos, as sequências de estados que passa um objeto e quais as iterações que estimulam essas mudanças.
Sequência	Permite representar como objetos do sistema interagem entre si para fornecer a funcionalidade do caso de uso.
Atividade	Pode ser usada para descrever cada um dos casos de uso; especifica o fluxo de controlo e as ações dos objetos para a realização de uma atividade relevante num sistema.
Objeto	Permite representar um diagrama de classe com um exemplo concreto, isto é, representa conjuntos de objetos, instancia e mostra um exemplo de iterações entre eles (um snapshot).

Tabela 2.1: Diagramas do UML

A conjugação destes diagramas permite que se aborde melhor a modelação dos sistemas, pois, cada um deles possibilita uma vista do sistema, realçando os aspetos importantes.

2.3 Ferramentas UML

Como já vimos, o UML é independente da linguagem de programação e das ferramentas de modelagem. Não é uma metodologia de desenvolvimento, sendo apenas uma

Diagrama De	Descrição
Comunicação	Nova designação para diagramas de colaboração onde explica a troca de mensagens entre objetos.
Componentes	Modela como os componentes do *software* estão organizados e quais são os seus relacionamentos.
Estruturas Compostas	Detalha de forma mais completa a informação de estruturas de uma classe ou componente.
Iteração	É uma variante do diagrama de atividade, permite que exista uma hierarquia entre diagramas. Captura o comportamento entre objetos dentro de um único caso de uso.
Pacote	Mostra como os elementos dos modelos são organizados em associações lógicas, pacotes, e as dependências entre esses pacotes.
Implementação	Permite descrever a arquitetura física do sistema, detalhando os nós, ambientes e *middleware*.
Temporal	Descreve a forma como uma entidade muda através do tempo, tipicamente como resposta a estímulos com base na noção de tempo.

Tabela 2.2: Novas diagramas do UML 2.0

linguagem normalizada para a construção de modelos de sistemas *software*. Contudo, a escolha da ferramenta a utilizar é importante para facilitar a modelagem de diagramas, melhorar a qualidade do *software* a ser desenvolvido e aumentar a produtividade na modelagem do sistema.

Existem atualmente centenas de ferramentas de desenho ou modelagem, tanto comerciais como académicas, para edição de diagramas UML [17] [35]. Estas ferramentas apresentam algumas aplicações informáticas (*debuggers*, aplicações de gestão de projetos, sistemas de críticas e outros) com o objetivo de aumentar a produtividade e automatizar, o melhor possível, as tarefas dos arquitetos e programadores de *software*.

A seguir é apresentado um estudo sobre estas ferramentas. Para o efeito, foi agrupado nas seguintes três categorias: **comerciais, não comerciais e *web***. Cada categoria apresenta uma tabela com as principais ferramentas e informações sobre as mesmas:

- **Nome (desenvolvedor)** – o nome da ferramenta em estudo e o nome da empresa ou pessoa que a desenvolveu entre parênteses.
- **Linguagem de implementação** – *Linguag. Impe* a linguagem de programação que foi utilizada para o desenvolvimento da ferramenta.
- **Plataforma**- as plataformas nas quais é possível trabalhar com a ferramenta.

- ***Open Source*** – se a ferramenta é *open source* ou não.
- **UML 2**- verifica se a ferramenta suporta o UML 2.0
- **XMI** – verifica se a ferramenta disponibiliza o diagrama editado em formato XMI ou importa nesse formato.
- **Geração de código** – *Ger. Cod.* verifica a possibilidade de geração de código para linguagens de programação (Java, PHP, C++, etc).
- **Reverse Engineering** – *Rever. Eng.* cria modelos a partir de código fontes (Java, PHP, C++, etc)

No apêndice A, na tabelas A.2 é apresentado mais comparações entre as ferramentas e na tabela A.1 uma comparação entre os tipos de diagramas UML suportados por estas ferramentas.

2.3.1 Ferramentas UML comerciaos

Grande parte das ferramentas de modelagem de sistema UML são comerciais. São ferramentas desenvolvidas para serem comercializadas ou com interesses empresariais. Pelo facto de serem desenvolvidas para as empresas que normalmente as utilizam para a modelação de sistemas complexos, apresentam mais recursos e funcionalidades. Exemplos destes recursos são: validação do diagrama – ajudar os desenvolvedores a detetar e corrigir facilmente os erros; geração do código – extrair código fonte a partir do modelo do sistema; *reverse engineering* — criar modelos a partir do código fonte, suporte a XMI, controlo de versões e gestão de projeto. Caraterizam-se por serem multiplataformas e suportarem em grande parte a modelação ou desenho dos diagramas das novas versões da linguagem UML.

Como normalmente são usados para modelagem de sistemas complexos em que trabalham um grande número de pessoas, apresentam um sistema de validação e verificação mais completo do que as ferramentas livres. Isto significa que há uma maior verificação das regras semânticas estáticas definidas pela especificação da UML e melhor verificação de consistência nos diagramas.

A maioria destas ferramentas (como por exemplo, o MagicDraw [30]) apresenta vários moldes (*templates*) para a criação de novos modelos, onde cada um dos quais disponibiliza um conjunto de elementos necessários para a modelação de um certo tipo

de sistema. Outra das funcionalidades que podemos encontrar nestas ferramentas é a geração de documentos em diversos formatos, como por exemplo, documentos *Microsoft Office (Word, Excel, PowerPoint).*

São poucas as ferramentas UML comerciais exclusivas para modelagem ou desenho de diagramas UML (StarUML [25]). A maioria são ferramentas visuais definidas para modelagem em vários domínios (modelagem de processos de negócios, modelagem computacional, arquitetura de dados, *software* de modelagem de simulação AnyLogic, entre outros), que dão suporte a linguagem UML (MagicDraw [30], UModel [2], Glifty [15]).

Ferramenta (*developer*)	Linguag. Imple.	Plataforma	Open Source	UML 2.0	XMI	Ger. Cod.	Rever. Eng.
MagicDraw (No Magic)	Java	Cross-platform	Não	Sim	Sim	Sim	Sim
Modelio (Modelio-soft)	Java	Windows, Linux, OS X	Sim	Sim	Sim	Sim	Sim
StarUML (MKLab)	Delphi	Windows, Linux, OS X	Não	Sim	Sim	Sim	Sim
Visio (Microsoft)	C++	Windows	Não	Sim	Sim	Sim	Sim
Rational Rose XDE (IBM)	Java	Windows, Linux, Unix	Não	Sim	Sim	Sim	Sim
UModel (Altova)	Java C# VB	Windows	Não	Sim	Sim	Sim	Sim

Tabela 2.3: Ferramentas do UML Comerciais

2.3.2 Ferramentas UML livres

Ferramentas UML livres (não comerciais) são normalmente desenvolvidas no meio académico. Por ter um âmbito mais pedagógico, geralmente apresentam menos recursos e funcionalidades que as ferramentas comerciais. Contudo, em geral, são ferramentas desenvolvidas para o domínio da modelagem de diagramas UML e apresentam modelos seguindo fielmente a especificação UML. Existem excelentes ferramentas UML livres (ArgoUML, Umbrello [44], Dia [10]) e com funcionalidades e recursos diferenciadores das outras ferramentas, como por exemplo, os sistemas de crítica *built-in* no ArgoUML [43].

Caracterizam-se por serem ferramentas simples com interface intuitiva e auto-explicativa. Apresentam facilidades no uso, pois grande parte destas ferramentas são usadas no meio académico para o ensino da linguagem UML. Por outras palavras, são usadas para modelar diagramas simples por pessoas (alunos) com pouca experiência e capacidade no domínio. São quase sempre *Open Source* para que possa ser usado, copiado, estudado, modificado e redistribuído.

A maioria destas ferramentas são multiplataformas e exportam e importam diagramas para o formato *XMI*. No entanto, ao contrário das ferramentas comerciais, nem sempre estas ferramentas suportam a modelagem ou desenho das novas versões da linguagem UML, conforme é apresentado na tabela A.1. Apesar disso, suportam grande parte dos diagramas e elementos considerados essenciais definidos pela especificação da UML.

Ferramenta (*developer*)	Linguag. Imple.	Plataforma	Open Source	UML 2.0	XMI	Ger. Cod.	Rever. Eng.
ArgoUML (Tigris.org)	Java	Cross-platform	Sim	Não	Sim	Sim	Sim
Dia (Alexander Larsson)	C	Cross-platform	Sim	Parte	Não	Sim	Não
Creately (Cinergix, Pty Ltd)	Adobe's Flex/ Flash	Web, Cross-platform	Não	Sim	Não	Não	Não
PlantUML (Arnaud Roques)	Java	Cross-platform	Sim	Sim	Sim	Sim	Sim
Umbrello (Umbrello Team)	C++ KDE	Unix-like, Windows	Sim	Não	Sim	Sim	Sim
UML Designer (Obeo)	Java, Sirius	Plugins Eclipse	Sim	Sim	Sim	Sim	Sim

Tabela 2.4: Ferramentas do UML livres

2.3.3 Ferramentas UML web

Consideramos como ferramentas UML *web*, as ferramentas UML projetadas para utilização através de um *browser* e através da Internet. Estas ferramentas normalmente permitem a colaboração de vários utilizadores na edição de diagramas em tempo real, com funcionalidades específicas que facilitam esse modo de edição. Exemplos destas funcionalidades são o *chat* embutido e o controlo de versões (*backup*). Para esses tipos

de ferramentas, normalmente é obrigatório ter uma conta.

As ferramentas para edição de diagramas UML na *web* normalmente são ferramentas para o desenho de diagramas em vários domínios que dão suporte ao desenho de diagramas UML, ou seja, não são editores exclusivos para UML. Caracterizam-se por serem ferramentas comerciais com poucos recursos e funcionalidades (não apresentam um sistema de validação, geração de código ou suporte a XMI).

Pelo facto de serem ferramentas *web*, apresentam algumas funcionalidades caraterísticas da *web*, como a partilha de diagramas nas redes sociais e páginas *web*. Exemplos de ferramentas que apresentam essas funcionalidades são: Cacoo [31], Lucidchart [18], Gliffy [15].

Ferramenta (*developer*)	Linguag. Imple.	Plataforma	Open Source	UML 2.0	XMI	Ger. Cod.	Rever. Eng.
Cacoo (Nulab, Inc)	HTML5	Web	Não	Sim	Não	Não	Não
Creately (Cinergix, Pty Ltd)	Adobe's Flex/ Flash	Web, Cross-platform	Não	Sim	Não	Não	Não
Gliffy (Gliffy)	HTML5 e Javascript	Web, (Browsers)	Não	Sim	Não	Não	Não
Lucidchart (Lucid Software)	HTML5 e Javascript	Web, (Browsers)	Não	Sim	Não	Não	Não

Tabela 2.5: Ferramentas do UML para web

2.4 Sistema de crítica

Um sistema de crítica é um mecanismo que atua sobre as ferramentas de modelagem de modo a fornecer correções e sugestões sobre os modelos a serem projetados [36]. Esses mecanismos são válidos, não só para verificação de construção sintática de uma linguagem de modelagem, mas também como heurística de bons projetos, suporte a tomada de decisões e verificação de consistência entre múltiplas visões em vários modelos dentro de um domínio.

Existem dois tipos de sistemas de crítica: autoritária e informativa. Sistema de crítica autoritária analisa o modelo de acordo com a presença e ausência de determinada

propriedade; e sistema de crítica informativa visa detetar potenciais problemas e auxiliar, tanto na resolução desses mesmos problemas, como na evolução do projeto [36].

Vários aspetos motivam a utilização de um sistema de crítica na modelagem de sistemas, como conhecimento limitado e pouca experiência no domínio por quem está a modelar os projetos, baixo custo de revisão imediata, redução do tempo de desenvolvimento, melhor gerência de riscos e possibilidade de aprendizagem contínua. Estes tipos de sistema atuam na descoberta antecipada de erros, apresentam as sugestões de melhororias e previnem erros críticos. Deste modo, estes sistemas embutidos nas ferramentas de modelagem ajudam na eficiência do trabalho, aumentam a produtividade e diminuem o custo do projeto, já que muitos dos erros vão ser detetados na fase de modelagem dos projetos.

Muitas pesquisas têm sido feitas no desenvolvimento das ferramentas de crítica [1, 26, 42]. Essas ferramentas de crítica são desenvolvidas com base no seu contexto e finalidade. ArgoUML [43], ArchStudio5 [9], SoftArch [16], DAISY [6], IDEA [4], ABCDE-Critic [41] e AIR [23] são todos exemplos dessas ferramentas. A ferramenta ArgoUML [43] possui um sistema de crítica *built-in*, que promove revisão discreta e transparente, onde apresenta possíveis melhorias durante a edição de diagramas UML. Estas críticas incluem erros sintáticos, guias de estudos e recomendações de desenvolvedores experientes. ArchStudio5 [9] e SoftArch [16] apresentam um sistema de crítica que monitoriza problemas em modelagem de arquitetura *software*. DAISY(*Domain and Application engineering using Integrated critiquing SYstems*) [6] usa três diferentes sistemas de crítica para apoiar a inconsistência de modelos e deteção de erros em modelo de engenharia de *software*. O IDEA (*Interactive DEsign Assistant*) [4] produz críticas com base no *design pattern* para melhorar a modelagem em UML. Com base em anotações, ABCDE (*Annotation Based Cooperative Diagram Editor*) [41] utiliza três sistemas críticos para apoiar as atividades de engenharia de domínio e engenharia de aplicações. Por fim, a ferramenta AIR (*Advisor for Intelligent Reuse*) [23] utiliza sistema de crítica para assistência de forma inteligente na engenharia de requisitos (*requirements engineering*).

Como podemos verificar, as ferramentas de crítica são usadas em vários domínios. Contudo, essas ferramentas produzem críticas para um problema específico num domínio. No nosso caso, vamos desenvolver este tipo de sistema como parte da ferramenta Kora.

Este sistema irá apresentar feedback visual no editor Eshu e textual no componente Kora para auxiliar os alunos na aprendizagem de modelagem de linguagens dia-

gramática, tais como UML e ERR . Este gerador de feedback deve ser extensível a novo tipos de diagramas.

Capítulo 3

Trabalho Prévio

Neste capítulo serão apresentados os mecanismos já desenvolvidos e as modificações que iremos fazer de modo a desenvolver o ambiente de aprendizagem da linguagem UML. Estes mecanismos são: Eshu [21], Enki [34] e Avaliador de diagrama [40].

3.1 Eshu - Editor de diagrama

Um editor de diagrama é um componente fundamental para o ambiente das linguagens diagramáticas. Contudo diferente dos editores de código para linguagens de programação existe uma carência no que consiste a editores e bibliotecas embutidas em aplicações *web* para as linguagens diagramáticas. De modo a contribuir para resolução desse problema, propomos o desenvolvimento do Eshu, um editor de diagrama *web* extensível e embutido em aplicações *web*, que requerem iterações de diagramas, como ferramentas de modelagens e aplicações *web*. Na primeira parte do trabalho [21], desenvolvemos uma biblioteca *JavaScript* com uma API que suporta a sua integração com outros componentes, incluindo a importação/exportação de diagramas no formato JSON para linguagens EER (*extended entity–relationship*).

Algumas funcionalidades desenvolvidas no Eshu na primeira parte do trabalho:

- Editar um diagrama ERR.
- Avaliações de diagrama ERR.
- Funções da API que permite aplicar restrições da linguagem ERR no editor de diagrama (por exemplo, restrições nas ligações entre os nós).

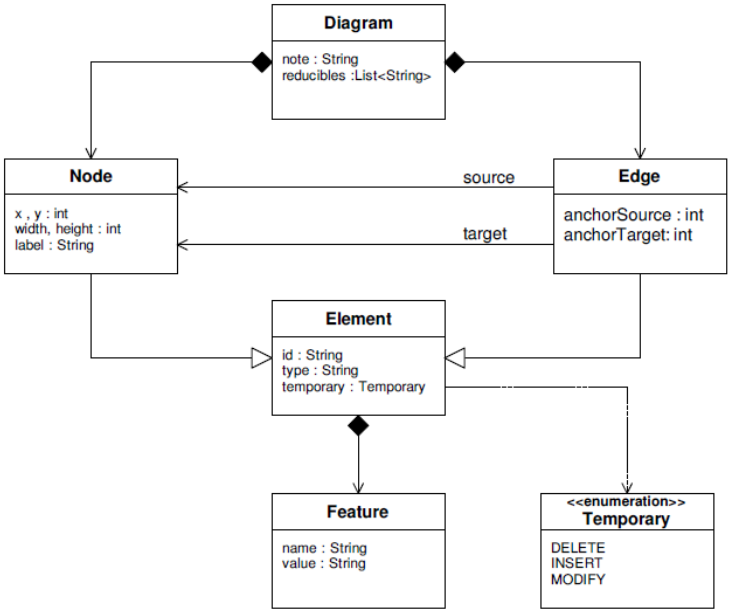


Figura 3.1: Diagrama de classe do esquema de dados usado pela API.

- Apresentação de *feedback* visual sobre as submissões de diagrama EER.

- Importar/exportar um diagrama em formato *JSON*.

O principal objetivo do Eshu é ser integrado em aplicações *web*. Por esse motivo ,a interoperabilidade é uma característica importante no seu *design*. Eshu tem uma extensa API que permite à aplicação *host* controla-lo. Alguns dos métodos mais importantes da API exigem uma serialização do diagrama, para importá-los ou exportá-los. Eshu usa sua própria serialização baseada em JSON, especificamente usando JSON-Schema [13]. A Figura 3.1 apresenta, em classes UML, o esquema de dados usado pela API.

Eshu tem definido na API alguns métodos básicos usados na parte da avaliação e apresentação de *feedback*. São eles setGraph() e getGraph() e importGraph(). O primeiro recebe como argumento um diagrama representativo, enquanto o segundo retorna um diagrama em formato JSON. O método importGraph() também recebe como argumento um diagrama, mas o diagrama contém apenas as diferenças para o diagrama atual, isto é, o *feedback* a ser inserido no diagrama.

3.1.1 Extensibilidade

Eshu foi projetado para ser extensível, para ser capaz de incorporar novas linguagens diagramáticas, com seus próprios nós e arestas, e impor restrições sobre as suas ligações [21]. Essas extensões exigem novas definições de *JavaScript* que têm de ser integradas com código-fonte Eshu. A seguir é apresentado um resumo sobre a extensibilidade do Eshu.

Para definir um novo nó (*Node*) é necessário passar como argumento a posição *x,y* e o *id* estender a classe *Vertice*, definir os *tipos do nó* a que se pode conectar e definir o *método draw*. A seguir é apresentado um exemplo para a criação do nó entidade do diagrama ERR.

```
1  function Entity (x , y , id ) {
2      Vertice.call (this , x , y , id ); // extend Vertice
3      this.type = " entity " ;
4      this.label = " ENTITY " ;
5      this.weak = false ;
6      this.cursorPosition = this . label . length -1;
7      this.listTypeCanBeconnected =[
8          " attribute " , " simpleNode " ,
9          " relationship " , " espGenCat " ];
10 }
```

Listing 3.1: Construtor Nó Entidade

E para definir uma aresta (**Edge**) é necessário passar como argumento o **nó de origem**, **nó de destino**, **hander do nó origem** e **hander do nó de destino**, **id** e **link** – link é o tipo de ligação –, **estender a classe Edge**. A seguir é apresentado um exemplo para uma aresta do diagrama ERR.

```
1      function EEREdge1 ( source , target , handleSource , handleTarget
           , id ) {
2          this . links = [
3              new SimpleNodeOther() ,
4              new EntityAttribute() ,
5              new EntityRelationship() ,
6              new AttributeAttribute() ,
7              new AttribuiteRelationship() ,
8              new RelationshipRelationship() ,
9              new EntityEspGenCat()];
10         Edge . call (this , source , target , handleSource ,
               handleTarget , id );
```

```
11 }
```

Listing 3.2: Definição de uma aresta.

Após criar os nós e as arestas, para definir o grafo é necessário criar a linguagem diagramática que vamos usar no grafo. Para isso, tem definido na API o método *setLanguage* que recebe como argumento o nome da linguagem, lista com o nome dos tipos de nós, lista com nomes de tipos de arestas que fazem parte da linguagem.

```
1     var graph = new Graph(div,700,400);
2         graph.setLanguage(Language( "err", ["attribute","entity","
              relationship","espGenCat"],
3         ["line","lineEGC"]);
```

Listing 3.3: Definição de uma aresta.

3.1.2 Validação

Para realizar a avaliação de usabilidade, o Eshu foi integrado num ambiente *web* pedagógico para o ensino de linguagens de programação – Enki [33] – para criar um módulo de avaliação automática de diagramas para a disciplina de Base de Dados da FCUP. Para a avaliação de diagramas foi utilizado um avaliador de grafos [40].

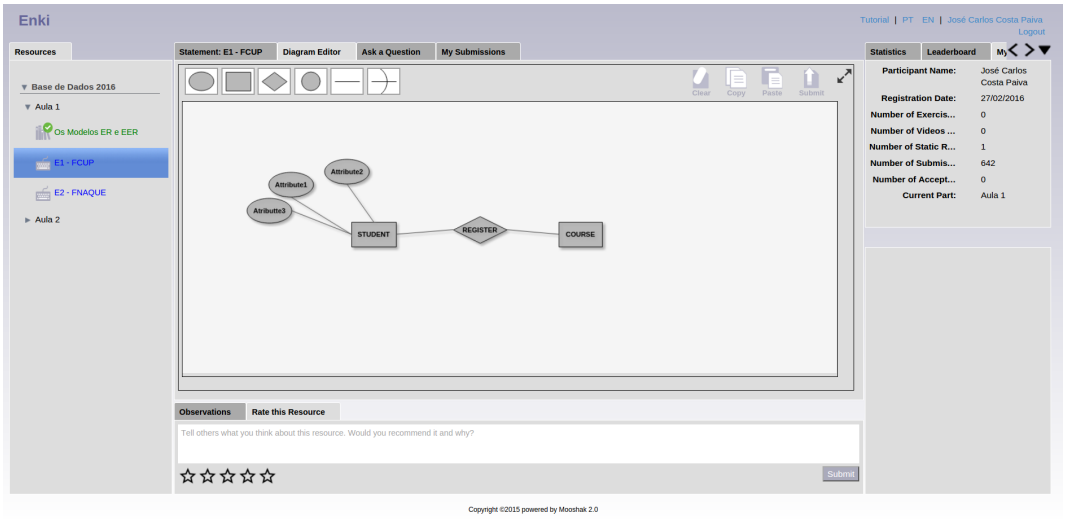


Figura 3.2: *Screenshot* do Eshu integrado no Enki

O Enki obtém o objeto *JSON* do diagrama atual (*getGraph*), guarda uma cópia numa variável e envia ao avaliador para avaliação. O avaliador compara com a solução e retorna as diferenças em formato *JSON*. De seguida, o Enki utiliza a função

(*importDiff*) da API do Eshu para apresentar o resultado, por cima do grafo corrente, com as diferenças enviadas pelo avaliador. Após examinar o *feedback*, o aluno clica no botão OK, sendo ocultado o diagrama que contém o *feedback* do avaliador e apresentado o diagrama submetido. Na Figura 3.3 é apresentada a comunicação do Eshu no caso da submissão de um exercício para o avaliador.

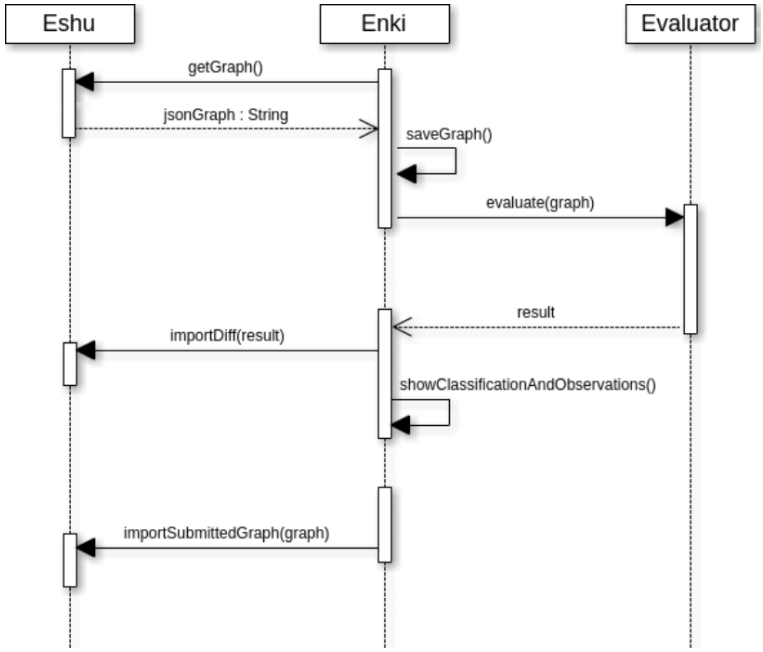


Figura 3.3: Comunicação do editor de diagrama Eshu com Enki sobre a submissão com o avaliador

Após o teste da usabilidade foi feito um questionário aos alunos no modelo de Nielsen's [28] usando um formulário do Google(apêndice E). Como pontos positivos foram identificados a compatibilidade, a consistência, a visibilidade e a facilidade de uso. Como pontos negativos, os estudantes identificaram a velocidade, a confiabilidade e a flexibilidade. Em geral, os alunos classificaram Eshu como uma ferramenta adequada (37,5%) e outros afirmaram que Eshu era muito bom ou bom (37,5%). Alguns alunos classificaram como mau ou muito mau (25%). A partir destes testes, foi possível identificar e corrigir algumas falhas e receber sugestões por parte dos alunos.

De modo a atingir os objetivos propostos inicialmente, iremos aplicar algumas alterações no Eshu, principalmente na parte de extensibilidade. Pretendemos estender o Eshu de modo a poder editar diagramas UML e apresentar *feedback* assíncrono sobre a linguagem UML através do sistema Kora. Também, pretende-se adicionar URL para informações externas em cada elemento do diagrama.

3.2 Enki- Ambiente *web* para aprendizagem de linguagens computacionais

Enki é um ambiente *web* integrado para aprendizagem de linguagens computacionais tais como linguagens de programação (*C, Java*) e linguagens de diagramas [33,34]. É uma GUI para a nova versão do *Mooshak* (versão 2.0), um *framework* para avaliação automática das linguagens informáticas, tanto a nível competitivo como pedagógico. Foi projetado para ser usado em diversas áreas de apoio ao ensino, como por exemplo, nas disciplinas de programação de ensino médio ou superior, nos cursos abertos *online*.

É uma versão simplificada de uma IDE, onde não é preciso instalar ferramentas complexas, como editores e compiladores, apenas acede-las no *browser* através da Internet. Não é preciso uma máquina com muitos recursos para utilizar a ferramenta, isto é, através de um simples *tablet* e através do *browser* é possível aceder ao Enki, o que seria virtualmente impossível para IDEs como Eclipse [19] ou NetBeans [27].

Enki é constituído por várias ferramentas inspiradas em mecanismos e ferramentas no ambiente de aprendizagem de linguagens de programação [34]. Essas ferramentas incluem um Serviço de Gamificação (*Gamification Service-GS*), em que utiliza técnicas caraterísticas de jogos (pontos, competição, progressão) para motivar e cativar a atenção dos alunos; um Serviço de Sequenciamento de Recursos (*Educational Resources Sequencing Service - ERSS*) para oferecer diferentes caminhos e recursos aos alunos de acordo com o seu conhecimento, objetivo e progresso; um Mecanismo de Avaliação (*Evaluator Engine-EE*) para dar *feedback* automático às soluções dos alunos; um Criador de Exercícios (*Exercise Creator-EC*) para permitir aos professores criarem exercícios; e um Repositório de Exercícios (*Learning Objects Repository -LOR*) para armazenar esses exercícios. Para além destas ferramentas, o Enki também promove a colaboração social e pode ser considerado como um sistema de aprendizagem com base nos Sistema de Gestão de Aprendizagem (*Learning Management System-LMS*).

A Interface foi desenvolvida para suportar diferentes resoluções e dispositivos e é constituído por várias janelas redimensionáveis que podem ser movidas de acordo com as necessidades e preferências dos alunos.

Os componentes : Eshu, Kora e o GraphEval serão integrados no Enki, de modo a implementar um ambiente de aprendizagem da linguagem UML, esse ambiente que terá como base a resolução de exercícios práticos sobre diagramas UML.

3.3 GraphEval - Avaliador de Diagramas

Os diagramas são representações esquemáticas de informação que, ignorando o posicionamento dos seus elementos, podem ser abstraídas em grafos. Com base nisso, Rúben Sousa e José Paulo Leal [40], criaram um algoritmo de comparação de grafos. Com base no algoritmo, desenvolveram um avaliador de exercícios baseados em diagramas para comparar dois diagramas (solução e tentativa), de forma a identificar as suas diferenças e calcular uma classificação. Com essa classificação e com esse conjunto de diferenças ajudar alunos com respostas erradas.

O algoritmo de comparação de grafos tem como principal função encontrar o mapeamento entre nós solução e tentativa que minimiza o conjunto de diferenças entre eles. Também pretende evitar o teste de todas as permutações possíveis com o objetivo de tornar o avaliador eficiente. Por isso, o algoritmo foi implementado de forma a testar em primeiro lugar os mapeamentos que possuem maior probabilidade de serem os corretos, permitindo, a certa altura, a aplicação de um critério de corte.

Para validar o funcionamento do algoritmo, foi criado um gerador de grafos conexos com parâmetros configuráveis. Foi possível criar pares de grafos próximos com um conjunto de diferenças bem determinado, o que permitiu executar milhares de testes, que validaram o funcionamento do algoritmo. Para além disso, foi criado um *parser* de forma a converter ficheiros com a representação dos diagramas em grafos estendidos. Este *parser* foi implementado para lidar com diagramas EER(*Extended Entity–Relationship*). Foi integrado no Mooshak [22] e usado como experiência na disciplina de Base de Dados do curso Ciências de Computação para avaliação de diagramas EER.

Com base nestes trabalhos, pretendemos implementar um ambiente de avaliação de linguagem de diagramas.

Capítulo 4

Kora

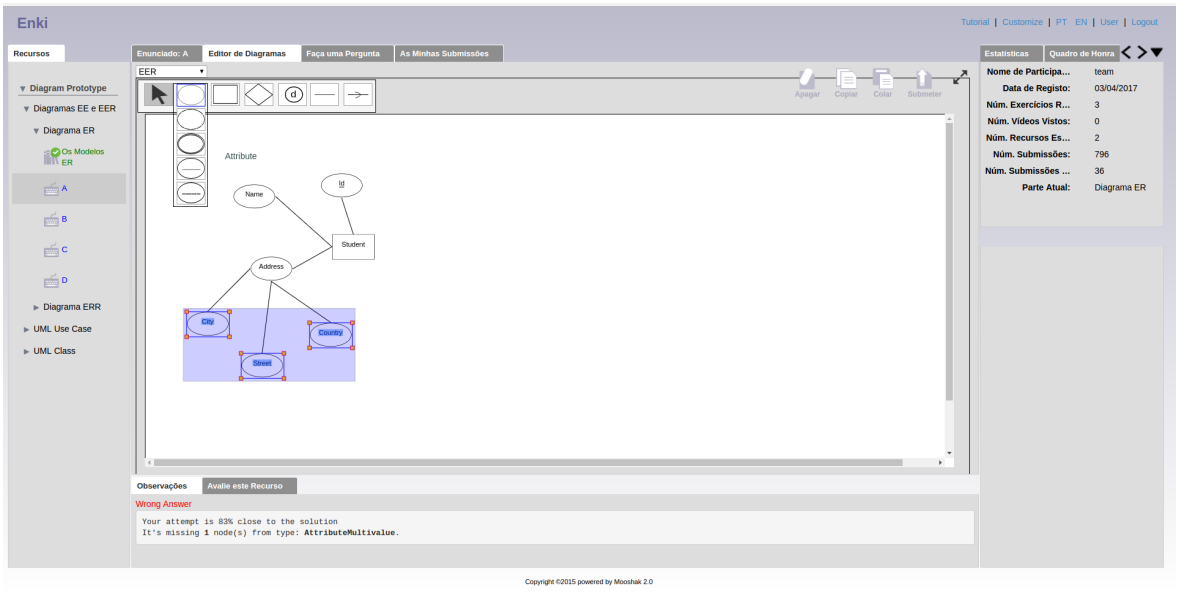


Figura 4.1: Interface do Kora integrado no Enki

O Mooshak é um sistema *web* para avaliação em ciências da computação que suporta avaliações de programas utilizadas em ciências de computadores. Foi desenvolvido e integrado no Mooshak um ambiente de avaliações de diagramas, suportado por um editor de diagramas e um avaliador de diagramas. Contudo, durante a validação do ambiente de avaliações de diagramas verificou-se uma série de deficiências, como a falta de suporte para múltiplas linguagens diagramáticas, *feedback*. Para colmatar essas deficiências foi projetado uma nova arquitetura, conforme é apresentado na secção 1.3, com introdução de novos componentes e modificações de alguns já existentes. Neste capítulo, é apresentado o componente Kora, que tem como objetivo executar

o editor Eshu, melhorar a qualidade de *feedback* gerado e dar suporte a múltiplas linguagens diagramáticas. Na figura 4.1 é apresentado um *screenshot* do ambiente de avaliações de diagramas no Mooshak.

4.1 Design do Kora

Figura 4.2: Diagrama de pacote do design do Kora

O componente Kora é dividido em duas partes, cliente e servidor. A parte cliente é integrada numa interface *web*, como é apresentado na Figura 4.1, e é responsável por executar o editor Eshu, bem como tratar das ações dos utilizadores e apresentar o *feedback* visual e textual. A parte do cliente é implementada seguindo o padrão de desenho *Model–View–Presenter* (MVP) para criação da interface gráfica. É constituído por três classes, `KoraPresenter`, `KoraViewImpl` e `FormatMessage` e uma interface `KoraView`. A interface `KoraView` define o modelo de dados que será exibido ou alterado na interface do Kora. A classe `KoraViewImpl` implementa a interface `KoraView`, exibe os dados (o modelo) e guia os comandos do utilizador (eventos) à classe `KoraPresenter` para atuar sobre esses dados. A classe `KoraPresenter` atua sobre a *Model* e a *View*. Ela recebe os dados do Kora e formata-os para exibi-los na *View*. A classe `FormatMessage` é responsável por converter o *feedback* e apresentar no Kora com base na linguagem selecionada (inglês e português).

A parte do servidor é responsável por avaliar diagramas, gerar *feedback* e disponibilizar serviços a serem usados pelo Kora cliente, como por exemplo as configurações do editor. É composta por quatro pacotes `Converter`, `Semantic`, `Sytaxe` e `Feedback`,

conforme é apresentado na figura 4.2. O pacote *Converter* contém as classes que convertem o diagrama de formato `JSON` para tipo grafo, a ser utilizado nas avaliações. O `Semantic` e `Sytaxe` contém as classes responsáveis pela avaliação de diagramas conforme é apresentado na secção seguinte. E por fim, o pacote `Feedback` contém as classes que gerem `feedback`.

4.2 Avaliação de diagramas no sistema Kora

Um diagrama é uma representação esquemática da informação. Essa representação está associada a elementos que têm certas características e um posicionamento no espaço. Abstraindo o *layout* (a posição dos elementos), os diagramas podem ser representados como grafos. A abordagem que se pretende seguir para a avaliação dos diagramas é a comparação de grafos. Assim, será possível analisar o conteúdo do diagrama sem dar relevância ao seu posicionamento ou formatação gráfica.

No Eshu 1.0 era feito a validação dos tipos de conexões durante a criação e edição, ou seja, se os nós de origem e de destino não puderam ser conectados, seria relatado imediatamente. No entanto, durante a validação do Eshu 1.0 [21], verificou-se que o editor ficava mais lento à medida que o número de nós aumentava, embora nem todos as verificações sintáticos fossem cobertas. Além disso, grafos sintaticamente incorretos estavam a causar problemas na geração de *feedback* pelo avaliador. Devido a estas questões, foi transferida a parte de verificação sintática para o Kora.

A avaliação do diagrama no sistema é dividida em duas partes: avaliação sintática e avaliação semântica, conforme são apresentadas nas subsecções seguintes.

4.2.1 Avaliação sintática

A avaliação sintática envolve a conversão do arquivo *JSON* numa estrutura de grafos e a validação da sintaxe da linguagem diagramática. Consiste em validar a organização estrutural da linguagem, com base no conjunto de regras definidas no arquivo de configuração. Nesta fase, são realizadas as seguintes tarefas: validação dos tipos para a linguagem; validação das arestas - para cada aresta é verificada se o tipo do nó origem e destino são válidos; validação dos nós - verifica se o grau de entrada e saída são válidos; validação do número de componentes conectados no diagrama; e validação da extensão - se a extensão do ficheiro que contém o diagrama tentativa e solução são

iguais, isto é, se pertencem à mesma linguagem.

4.2.2 Avaliação semântica

A avaliação semântica consiste em comparar os diagramas tentativa e solução, seguindo o algoritmo de avaliação gráfica [40]. O avaliador recebe um grafo como uma tentativa de resolver um problema e compara-o com um grafo solução, de modo a encontrar o melhor mapeamento dos nós da solução nos nós tentativa e, assim, minimizar o conjunto de diferenças e maximizar a classificação.

4.2.3 Avaliação do diagrama

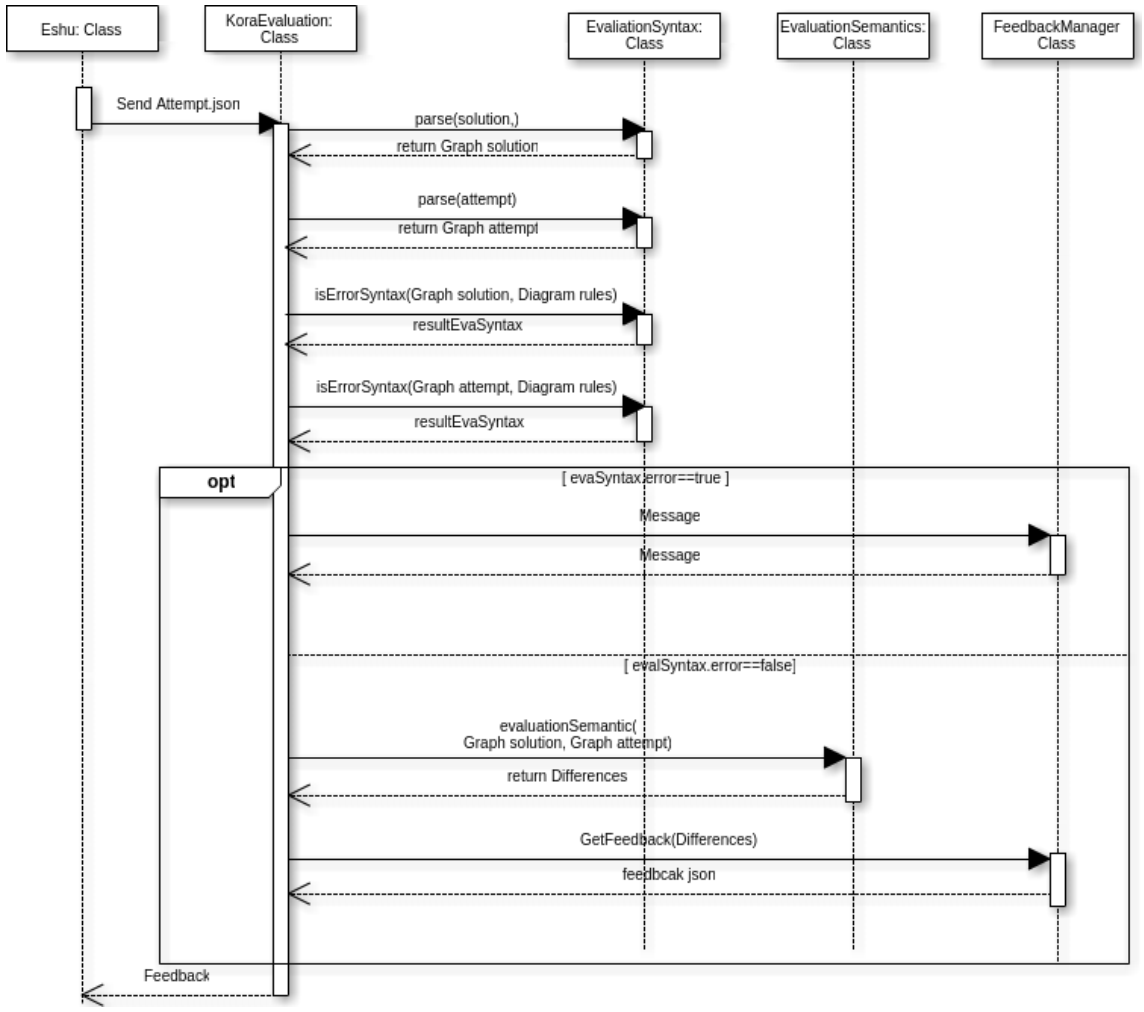


Figura 4.3: Diagrama de sequência da validação de diagrama no sistema Kora

A figura 4.3 apresenta o diagrama de sequência UML da avaliação do diagrama no sistema Kora. O cliente Kora obtém o grafo do diagrama no formato JSON através de uma função da API Eshu - `Eshu exportGraph()` e envia para o Kora Server (`KoraEvaluation`). O `KoraEvaluation` chama a classe `EvaluationSyntax` que faz o *parsing* das informações do JSON da solução e da tentativa, reunindo as informações necessárias sob a forma de uma estrutura de grafos, para representa-las nas próximas avaliações desses diagramas. A seguir, a classe `EvaluationSyntax` verifica a existência de qualquer erro de sintaxe, abortando a avaliação caso detete algum erro. Se não haver nenhum erro de sintaxe, passa-se para a avaliação semântica (*EvaluationSemantic*). Nesta fase, o avaliador recebe dois grafos: o grafo de tentativa e um grafo de solução. Na comparação dos grafos são localizadas os erros inseridos nas listas de diferenças. Com base nessa lista, é gerado o respetivo *feedback* (FeedbackMenager) e calculada uma classificação, de modo que o diagrama possa ser melhorado.

4.3 Geração de feedback

Como foi referido na secção anterior, a avaliação semântica fornecida pelo `Kora` baseia-se nas diferenças dos grafos (tentativa e solução) calculadas pelo avaliador de grafo. O avaliador de grafos compara os dois grafos, retorna um conjunto de diferenças e com base nessas diferenças, é gerado o *feedback* que é apresentado no `Enki` em forma de texto e também, quando for necessário, em formato visual no Eshu. No entanto, quando a tentativa do aluno estiver longe da solução, são geradas muitas diferenças.

Para lidar com esse problema, foi desenvolvido um gerador de *feedback* incremental, ao qual chamamos `Kora` (FeedbackManager - secção seguinte) . Este gerador usa várias estratégias de modo a agregar uma lista de diferenças numa única mensagem com um princípio simples: a mensagem mais geral que ainda não foi apresentada ao utilizador é selecionada como *feedback*. Por exemplo, várias diferenças que reportam a falta de nós do mesmo tipo podem ser agrupadas numa única mensagem "Faltam n nós do tipo T". Uma estratégia ainda mais detalhada pode ser mostrar o nó que falta no diagrama. Uma estratégia específica pode não ser aplicável a alguma lista de diferenças. Neste caso, nenhuma mensagem é produzida.

A lista de mensagens geradas é classificada de acordo com a generalidade. As mensagens gerais têm precedência sobre mensagens específicas. No entanto, se uma mensagem já foi enviada como *feedback* não é repetida. A mensagem a seguir é enviada. Usando essa abordagem, caso o aluno persiste no mesmo erro as mensagens com mais

detalhes são enviadas.

4.3.1 Arquitetura e funcionamento do FeedbackManager

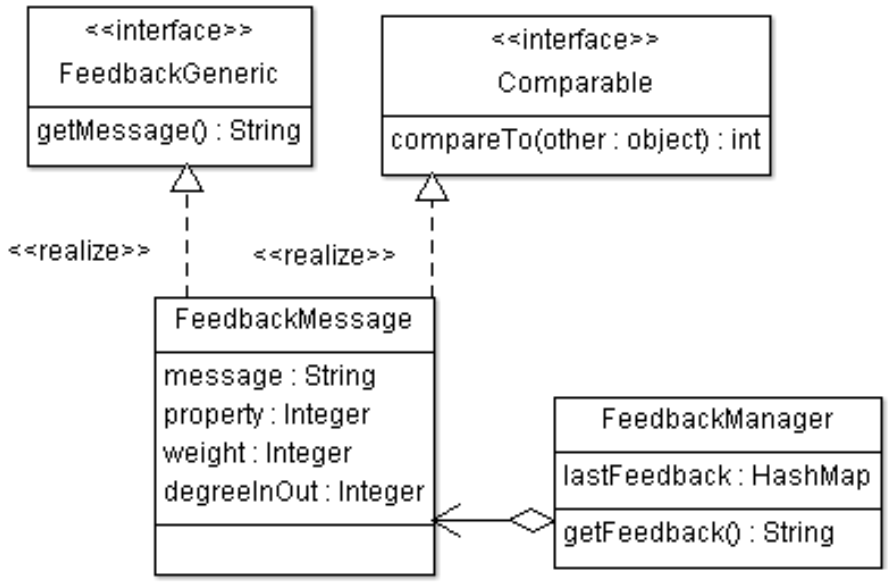


Figura 4.4: Diagrama de classe do *FeedbackManager*

`FeedbackManager` recebe uma lista de diferenças calculadas pelo avaliador de grafos (`GraphEval`) entre o grafo tentativa e o grafo solução. A partir desta lista é gerado um conjunto de `FeedbackMessage` sendo que cada `FeedbackMessage` é baseada numa diferença gerada. Cada `FeedbackMessage` contém um código da mensagem (formado pelo código da mensagem e os parâmetros a serem apresentados na mensagem), o número de propriedades apresentados no *feedback*, o peso definido para cada *feedback* e o número de entrada e saída, caso seja um nó. Após gerar os `FeedbackMessage`, a classe `FeedbackManager` ordena-os pela generalidade da mensagem, isto é, das que apresentam as mensagens mais gerais para as mais específicas, com base no número de propriedades, peso e grau de entrada/saída. Para além disso, remove os `FeedbackMessage` já enviados da lista. No fim, é selecionado o *FeedbackMessage* que estiver no topo da lista e é enviada a `mensagem` (código da mensagem e os parâmetros) ao Kora cliente. No Kora cliente, a mensagem é convertido para texto e é apresentado o respetivo texto, de acordo com a linguagem selecionada (português ou inglês).

4.3.2 Características do gerador de feedback

A seguir é apresentado algumas das principais características do sistema de geração de *feedback*.

- **Fornece feedback especializados em cada caso** – para cada tentativa Kora gera *feedback* diferente, *feedback* para inserir, eliminar ou modificar um nó, uma aresta ou uma propriedade do nó/arestas que fazem parte do diagrama.
- **Fornece feedback progressivo** – com o princípio de selecionar a mensagem mais geral, o gerador de *feedback* primeiro seleciona a mensagem que apresenta menos detalhes e vai acrescentando detalhes no *feedback*, caso o erro persista.
- **Controla a qualidade e quantidade do feedback** — após cada submissão o `KoraManger` controla a quantidade e qualidade do *feedback* a ser enviado. Isto é, seleciona o melhor *feedback* com base nas diferenças geradas pelo avaliador e o *feedback* já enviado.
- **Fornece feedback visual** – é apresentado o *feedback* visual quando já não é possível apresentar mais *feedback* textual, ou então no caso de ser *feedback* sobre erro sintático. No exemplo a seguir é apresentado um caso em que é aplicado o feedback visual.

4.3.3 Exemplo de feedback gerado pelo KoraManager

Na figura 4.5 é apresentado dois diagramas do tipo ER e as mensagens geradas pelo `KoraManager`. O diagrama A é solução e o diagrama B a tentativa.

```
1ªtentativa
Your attempt is 36% close to the solution
It's missing 2 node(s) from type: Attribute.

2ªtentativa
Your attempt is 36% close to the solution
It's missing 1 node(s) from type: AttributeMultivalue.

3º tentativa
Your attempt is 36% close to the solution
```

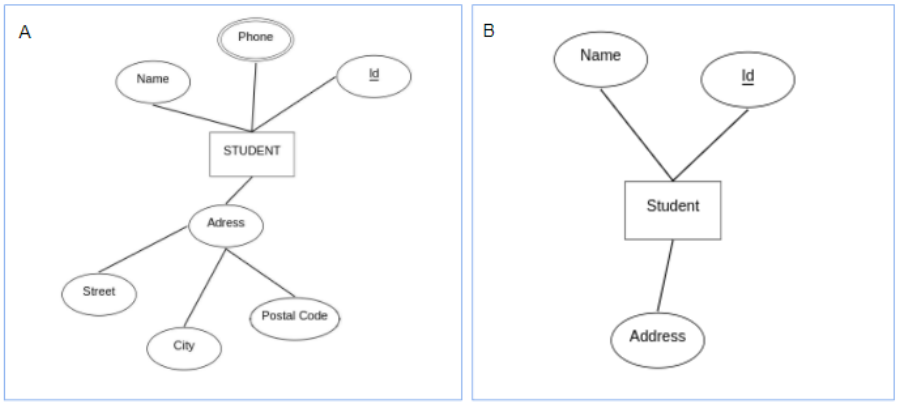


Figura 4.5: Exemplo de diagrama (ER)

```
It's missing 2 node(s) from type: Attribute.
Name suggestions: [ Street, City, Postal Code ]

4° tentaiva
Your attempt is 36% close to the solution
It's missing 1 node(s) from type: AttributeMultivalue.
Name suggestions: [Phone]

5° tentativa
Your attempt is 36% close to the solution
It's missing one node from type: Attribute.
Hint - Label [ Street, City, Postal Code ],
connected to other 1 node(s)
```

Comparando a solução (diagrama A) e a tentativa (diagrama B), podemos verificar que faltam três nós (dois do tipo `Atributo` e um do tipo `Atributo Multivalor`). Ao submeter o diagrama para a avaliação, a mensagem gerada pelo `KoraManger` na primeira tentativa é que faltam 2 nós do tipo Atributo. É selecionado o tipo `Atributo`, pois é o tipo de nó que apresenta maior número de nós em falta no diagrama tentativa (2). Na segunda tentativa, o `KoraManger` gera um *feedback* para o tipo `Atributo Multivalor` com o mesmo grau de detalhe. Já na terceira e quarta tentativa, o `KoraManger` acrescenta mais detalhes no *feedback*, sugerindo nomes para os nós, nomes esses que são dos nós da solução. Na quinta tentativa o `Kora` apresenta um *feedback* já especializado sobre um nó com todos os detalhes já apresentados e acrescenta ainda o grau de entrada e de saída do nó. Nesse caso, por ser um *feedback* especializado sobre

o nó, são também enviadas as informações que permitem apresentar o *feedback* visual no Eshu.

Na figura 4.6 são ilustrados dois exemplos de *feeback* visual apresentado no Eshu pelo Kora. No diagrama C, o diagrama contém um nó a mais (nome do nó *phone*) e esse nó é destacado de modo a ser eliminado. No exemplo D, é um caso em que falta um nó e é apresentado ao utilizador esse nó de modo a ser inserido.

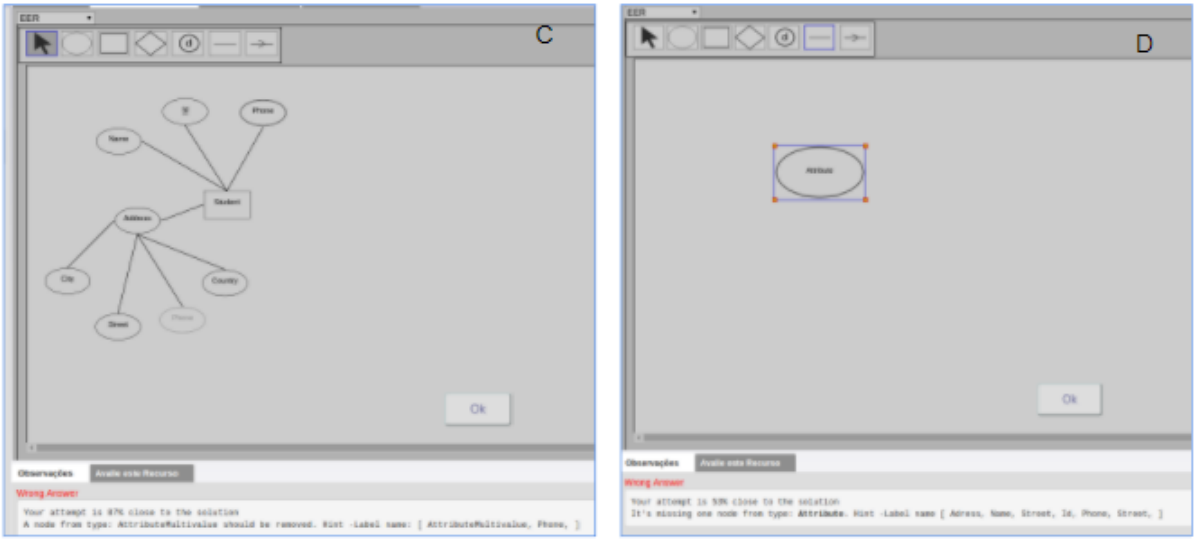


Figura 4.6: Exemplo de feedback visual no diagramas

4.4 Resumo

Neste capitulo foi abordado o componente Kora, que foi introduzido no Mooshak com o objetivo de melhorar a qualidade de *feedback* gerado e dar suporte a múltiplas linguagens diagramáticas. O *design* do Kora está dividido em duas partes, Kora `Client` e Kora `Server`. Na parte do Kora `Client` é executado o editor de diagrama Eshu que é apresentado no capítulo seguinte. A parte do Kora `Server` recebe o JSON do diagrama do editor e converte-o para grafos, que processa a avaliação sintaxe utilizando o avaliador de grafo (GraphEval). Além disso, Kora gere os *feedbacks* (textual e visual) a serem apresentados no Kora `Client`.

Como já foi referido, um dos objetivos na introdução do Kora no ambiente de avaliação de diagrama do Mooshak consiste em melhorar a qualidade do *feedback*. Para isso, o Kora apresenta um *feedback* especializado em cada caso; fornece um *feedback*

progressivo, controla a qualidade e a quantidade do *feedback* apresentado a cada aluno após uma submissão; quando possível, integra o *feedback* no editor de diagramas que identifica os nós e arestas no próprio editor que contém erros.

Capítulo 5

Editor de diagrama - Eshu

Um diagrama é composto por um conjunto de `nós` e `arestas`. Cada `nó` tem uma posição e uma dimensão e cada `aresta` liga um nó origem a um nó destino. O Eshu 2.0, de forma semelhante ao Eshu 1.0 [21], segue uma abordagem orientada a objetos para *Javascript*, mas separa a parte de dados da parte de visualização e edição. Foi desenvolvido em *Javascript* com base no elemento *canvas* do *HTML5* e integrado no Kora *client* para criar o editor de diagrama. Este capitulo apresenta o novo *design* e implementação do Eshu, a parte da Interface e Extensibilidade.

5.1 Design e Implementação

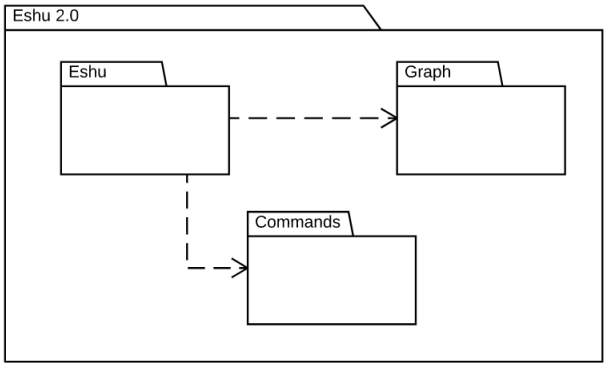


Figura 5.1: Diagrama de pacote do Eshu

Eshu 2.0 é constituído por três pacotes: `eshu`, `graph` e `commands`. O pacote `graph` tem as classes responsáveis pela criação de nós, arestas, classes responsáveis pelo armazenando e operações das informações do diagrama (`Quadtree`, inserir, remo-

ver, salvar alterações e selecionar um elemento). O pacote `eshu` contém as classes responsáveis pela interface do utilizador, incluindo manipuladores para interação do utilizador, métodos para exportar e importar o diagrama no formato JSON, métodos para apresentar *feedback* visual no editor de diagramas, entre muitos outros. O pacote `commands` contém as classes responsáveis pela implementação das operações, como desfazer, refazer, colar, remover ou redimensionar.

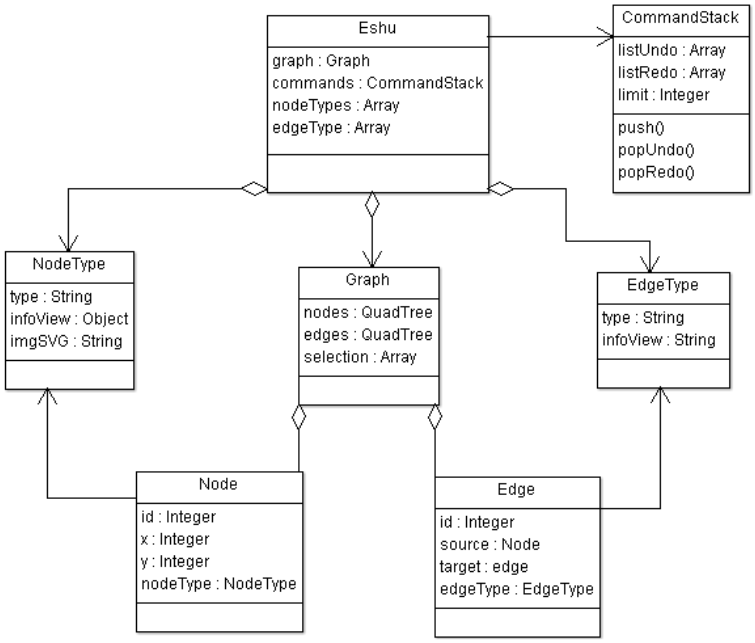


Figura 5.2: Diagrama Classe do Eshu

5.1.1 Pacote Graph

O pacote `graph` contém as classes responsáveis pelo armazenamento, criação, alteração e remoção de nós e arestas que constituem o diagrama. Neste pacote encontram-se as classes`Graph`, `Quadtree`, `Vertice`, `Edge`, `TextBox` e `RectangleSelection`.

5.1.1.1 Classe Graph

A classe `Graph` representa um grafo constituído por um conjunto de nós e arestas, e por funções que permitem operar sobre eles. Estas funções permitem inserir, modificar, selecionar ou remover um nó ou uma aresta do grafo, e também, exportar o grafo para uma estrutura JSON ou importar da estrutura *JSON* para grafo. Existem ainda

funções para copiar, cortar e colar um nó ou um conjunto de nós. Os nós e arestas são guardados numa estrutura do tipo textttQuadree.

5.1.1.2 Classe Vertice

No Eshu, um nó é representado por uma estrutura chamada `Vertice`. Esta estrutura contém um identificador, campos e funções, que permitem posicionar e atualizar a posição do nó e uma etiqueta do tipo `TextBox`, que é utilizada, normalmente, para editar o nome do respetivo nó. Cada `Vertice` contém também um `nodetype` com informações gerais para o tipo de nó, tais como o nome do tipo do nó, a imagem SVG, as configurações para a *label*, altura e largura padrão ou ainda o *url* (*link*) que é apresentado na janela de propriedades com informações sobre o nó. O `Vertice` apresenta uma lista de `Box`, chamadas de `handlers`, que são os pontos do nó onde se podem selecionar e redimensionar o nó. Contém também uma outra lista de `Box`, chamada de `anchors`, que guarda e atualiza os pontos do nó em que se podem ligar uma aresta.

Além da estrutura `Vertice`, definimos também um tipo `ComplexVertice`, que estende o tipo `Vertice`, e contém uma lista de `Container` B.8. Cada `Container` é um contentor que pode ter várias *labels*. Um exemplo de um nó que é representado pelo tipo `ComplexVertice` é o nó classe no diagrama UML de classes, em que o nome da classe é editado na *label name* do nó. O elemento atributo é um `Container` onde podemos adicionar e remover *labels* para editar elementos atributo. Ao contrário do `Vertice`, no `ComplexVertice` não é utilizado a imagem SVG do tipo de nó, mas sim, é desenhado um retângulo (pode ter cantos arredondados), e adicionada uma linha que separa cada`Container`.

Outra característica importante dos nós no Eshu é a possibilidade de definir ligações entre os nós por sobreposição, por exemplo, se um nó A sobrepor o nó B, é definido uma ligação do tipo `include` do nó A para o no B. Este mecanismo de ligação é importante para poder definir nós como `package` no diagrama de pacote, `RectangleContainer` e `swimlanes`.

5.1.1.3 Classe Edge

As arestas são consideradas as uniões entre os vértices. No Eshu uma aresta é representada pelo tipo `Edge` e recebe como parâmetro dois nós – um `source` (nó de

origem) e `target` (nó de destino) –, um `id` identificador da aresta e uma lista de pontos que servem como caminho da aresta. Além destes elementos, uma aresta contém um objeto `nodeType` com as informações gerais do nó, uma etiqueta para a aresta, uma lista de `handlers`, isto é, os pontos que definem o caminho da aresta. A classe `Edge` contém funções que permitem representar as arestas definidas no DL^2. Dentre essas funções destacamos o `draw` que desenha a aresta, `contains` verifica se a aresta contém um determinado ponto com as coordenadas x e y, `updateBorder`. A classe `Edge` tem definidos vários tipos de setas, que podem ser selecionadas na configuração DL^2 e representadas nas extremidades das arestas. Cada aresta é representada dentro de um retângulo, de modo a serem inseridas na `Quadtree`.

5.1.1.4 Classe Quadtree

Em geral, o número de nós e arestas num diagrama é bastante pequeno, uma vez que grandes diagramas são difíceis de entender. No entanto, pequeno é relativo e, para alguns computadores, algumas dezenas de nós e arestas podem ter impacto na eficiência, como foi analisado no Eshu 1.0 através das operações nas estruturas de dados (lista e quadtree). A operação mais exigente do ponto de vista da eficiência é a seleção de nós e arestas. Uma pesquisa por um nó ou aresta numa lista tem uma complexidade computacional de O(n), o que provou ser muito ineficiente. Por isso, definimos uma estrutura `quadtrees`, que, com uma complexidade de O(log(n)), permite uma pesquisa por um elemento de forma muito mais eficiente. Uma `quadtree` é uma estrutura de dados utilizada para codificar imagens, mas no Eshu a `quadtree` é utilizada para armazenar os nós e arestas, isto é, tendo como utilização principal o armazenamento de uma decomposição recursiva do espaço (área de desenho). A ideia fundamental é poder dividir a janela de diagrama em quatro quadrantes, sendo que cada quadrante pode ser dividido novamente em quatro sub-quadrantes, e assim sucessivamente. Na `quadtree`, a janela de diagrama é representada por um nó pai, enquanto que os quatro quadrantes são representados por quatro nós filho, em uma ordem pré-determinada. Sendo que, uma quadtree é um tipo especial de árvore onde todos os nós ou são nós folha ou têm quatro nós filho. A principal desvantagem do uso de `quadtree` para indexação de nós e arestas é que esta exige reindexá-los quando eles são movidos, mas o ganho de eficiência na pesquisa por elementos compensa.

5.1.1.5 Classe RectangleSelection

A maioria dos editores têm definidos uma ferramenta `RectangleSelection` que tem como funcionalidade selecionar regiões retangulares no editor. Para isso, no pacote `Grafo` encontra-se definido uma classe chamada `RectangleSelection` com as funções que permitem selecionar mover, copiar, cortar e colar um nó ou um conjunto de nós. Ao desenhar um rectângulo na área de desenho, todos os nós que intercetarem o retângulo são selecionados e é desenhado um retângulo à volta destes nós. Além disso, esta classe tem definida funções que permitem alinhar horizontalmente (esquerda, centro, direita), verticalmente (cima, meio e baixo) e equalibrar os nós selecionados. Esta classe permite definir uma ferramenta básica, mas muito usada na edição de diagramas, principalmente no alinhamento dos nós.

5.1.1.6 Classe TextBox

No Eshu, tanto o nó como aresta contém uma etiqueta (*label*) que permite editar texto (nome). Esta *label* é definida pela classe `TextBox` que define uma área no nó ou na aresta em que é possível editar texto (multilinhas) diretamente. Esta classe contém funções que permitem adicionar e remover um carácter, inserir quebra de linhas, selecionar *substrings*, entre outros. Através da linguagem de configuração (DL^2) é possível configurar o estilo do `TextBox`, com por exemplo o tamanho da fonte, o alinhamento do texto, cor do texto, entre outros, definir a posição da *label* no nó ou aresta e definir uma expressão regular para validar o texto da *label*.

5.1.2 Pacote Eshu

5.1.2.1 Classe Eshu

A classe `eshu` é responsável pela criação do editor do diagrama. Esta contém um elemento `canvas` que é a área de desenho do editor, um campo do tipo `graph` que guarda a representação garfo do diagrama, um objeto do tipo `commandStack` que permite executar os comandos *Undo* e *Redo*, um objeto do tipo `FormatPanel`, que permite criar a janela de propriedades do editor. Esta classe também é responsável por definir os métodos que permitem desenhar os elementos na área de desenho (*draw*). Contém funções que permitem criar os `NodeTypes` e `EgeTypes` definidos na DL^2 e duas listas que as armazena, uma lista que armazena os NodeType e outra lista

que armazena os `EdgeTypes`. O Eshu contém ainda um objeto do tipo `toolbar` que representa os `NodeTypes` e `EgeTypes` como opção a ser inserido no diagrama. Nesta classe também se encontra definido um campo de nome `graphState` que guarda o estado do grafo e é atualizado sempre que o estado muda. Como exemplo de estado temos o nó selecionado, arestas selecionadas, `rectangleSelect` selecionado, nó ou aresta selecionada na `toolbar`, entre outros. Este campo é importante para selecionar o comportamento do diagrama na classe `Events`.

5.1.2.2 Classe Events

Nesta classe são definidas todos os manipuladores dos eventos ocorridos na *canvas* que representa a área de desenho do `eshu`. Eventos esses que são responsáveis por controlar as ações do utilizador e definir um comportamento do editor quando se produzem, isto é, permite ao utilizador interagir com o editor de diagrama. Os eventos definidos nesta classe são `mousedown`, `mousemove`, `mouseup`, `keydown` e `keypress`. Os dois últimos são utlizados na edição de textos na *labels* e atalhos para alguns comandos como copiar, colar, cortar, undo, redo, selecionar todos os elementos e eliminar um elemento. Nesta classe estão definidas as operações que operam sobre o grafo que permitem selecionar, mover, remover um elemento na estrutura grafo e consequentemente no editor, redimensionar um nó e redefinir uma aresta. Cada evento definido tem um comportamento diferente de acordo com o estado do grafo (`graphState`). Por exemplo, o comportamento para ação do evento `mousedown` no diagrama pode inserir um nó, uma aresta, selecionar ou desselecionar um elemento no diagrama, pois esse comportamento depende do estado do grafo selecionado.

5.1.2.3 Classes Layout e Configuration

A interface do Eshu é constituída por uma área de desenho e um *toolbar*. Nesta classe encontram-se definidas as funções que definem a organização dos componentes que constituem a interface do diagrama. Funções essas que permitem posicionar o *toolbar* na vertical ou horizontal e redimensionar a *canvas* que define a área de desenho. Esta classe tem uma função que permite incrementar a largura ou altura da área de desenho, ao arrastar um nó para o canto direito o editor aumenta automaticamente a largura e ao arrastar para o canto inferior aumenta a altura. Nesta classe encontram-se definidas funções que permitem apresentar ou ocultar uma grelha (*grid*) na área de desenho.

No ficheiro configuração encontram-se definidas todas as funções que permitem confi-

gurar a parte do estilo do Eshu definido na DL^2, como o caso do cor de fundo, cor da linha da grelha, cor do fundo da *toolbar* e tamanho dos elementos na *toolbar*. Também encontram-se definidas funções para alterar o estilo de um elemento que são usados pelo `Objeto FormatPanel`, como por exemplo, o alinhamento de um conjunto de nós aumentar x e y (altura e largura) de um nó selecionado entre outros.

5.1.2.4 Classe FormatPanel

O elemento `FormatPanel` é uma janela de propriedades que apresenta vários campos que permitem alterar algumas configurações do estilo do editor de diagramas, como cor de fundo de uma grelha. Para além disso, permite editar elementos selecionados no diagrama. A janela pode ser apresentada, ocultada ou desativada e posiciona-se sempre no lado direito da área de desenho do diagrama. Na janela só são apresentados os campos necessários para editar o elemento selecionado. Foi implementado utilizando as funções da API do Eshu.

5.1.3 Pacote Command

Num editor de texto ou diagrama, durante a edição, é normal enganarmos-nos numa ação ou operação. Para colmatar esses erros é importante ter um mecanismo que permita voltar atrás (*undo*) ou avançar (*redo*). Por esse motivo, Eshu tem definido comandos `Desfazer` (*Undo*) e `Refazer` (*Redo*). O comando `Refazer` permite reverter a última alteração que foi feita no editor. Isto é, se um utilizador fizer uma alteração no diagrama e não deseja manter, o comando `Desfazer` permite-lhe desfazer a última alteração e retorne o diagrama ao seu estado anterior. A operação que foi "desfeita"não é perdida imediatamente: o utilizador pode recupera-la usando o comando `Refazer`, imediatamente. Isto é, o comando `Refazer` reverte os efeitos do comando `Desfazer`. Cada ação "Desfazer"pode ser revertida por uma única ação "Refazer". O utilizador pode alternar "Desfazer"e "Refazer"quantas vezes quiser, mas só pode aplicar uma operação "Refazer"se a última ação aplicada foi "Desfazer".

Para implementação destes comandos utilizamos o padrão de desenho `Command` [14]. O padrão `Command` permite encapsular um comando ou pedido num objeto. Assim, definimos uma interface `Command` e criamos cinco tipos de comandos que estende a interface `Command`: `Insert`, `Delete`, `Move`, `Paste` e `Resize`, onde cada uma implementa três métodos definidos na interface: `execute`, `undo` e `redo`. Também definimos uma estrutura `CommandStack` que guarda os históricos dos comandos executados e permite

chamar os métodos *undo* e *redo* dos comandos. `CommandStack` contém duas listas, uma lista que guarda os comandos *undo* e outra que guarda os comandos *redo*. Ao aplicar uma dessas ações no editor é criado um objeto do tipo de ação e é inserido na estrutura `CommandStack` do editor na lista do *undo*. No Eshu, o comando *redo* e *undo* pode ser multinível ou único nível, e é possível definir o número de níveis para os comandos na linguagem de configuração DL^2.

5.2 Interface

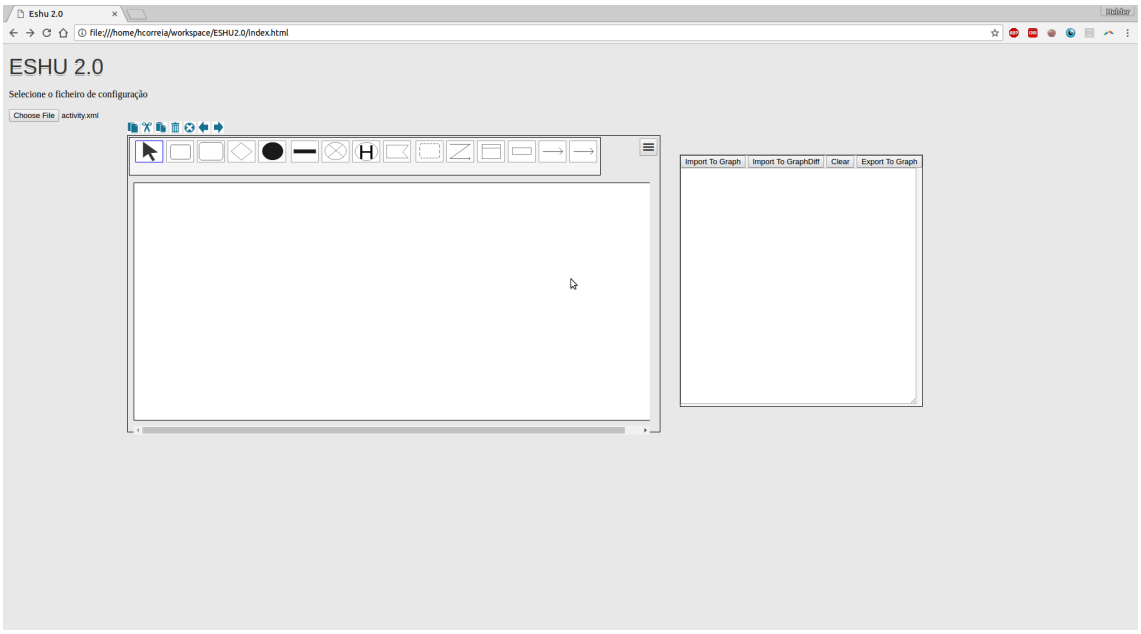


Figura 5.3: Screenshot da aplicação teste do Eshu

Eshu não é uma ferramenta independente, mas sim uma ferramenta que deve ser ligada a um sistema mais complexo. A interface de utilizador do Eshu aparece no contexto de uma aplicação *host*. Durante o desenvolvimento de Eshu, foi desenvolvida uma aplicação de teste para simular uma aplicação *host*. A Figura 5.3 apresenta um *screenshot* da aplicação de teste. Na verdade, a interface do Eshu é apenas o retângulo que inclui a área de desenho e a barra de ferramentas para selecionar nós e arestas. Todos os outros *widgets*, como os botões e a caixa de texto, fazem parte da aplicação teste. Esses *widgets* interagem com o Eshu através das funções definidas na API do Eshu, de forma semelhante a uma outra aplicação *host*. Por exemplo, a caixa de texto ao lado da interface do Eshu, é utilizada para importar e exportar um grafo em formato JSON.

Eshu é iniciado pelo *host*. Ao iniciar, o Eshu tem uma certa largura e altura que, por padrão, é o tamanho da área de desenho de diagrama. Contudo, se o utilizador arrastar um elemento ou conjunto de elementos para a direita ou para baixo, a área de desenho será aumentada automaticamente. No entanto, o tamanho do Eshu permanecerá o mesmo e são adicionadas *scroll bars* de modo a permitir a navegação na área de desenho do diagrama ampliado.

Os tipos de elementos do diagrama, nós e arestas, disponíveis para a linguagem diagramática corrente são apresentados na barra de ferramentas, logo acima da tela. Como foi referido anteriormente, este painel é parte integrante da Eshu. A barra de ferramentas pode ser posicionada verticalmente ou horizontalmente, ou ainda pode ser ocultada, usando as funções definidas na API do Eshu. Depois de selecionar um tipo de elemento, o utilizador pode inserir uma nova instância do elemento em qualquer lugar na área do desenho de diagrama e pode até se sobrepor a outros elementos. Contudo, uma aresta para ser inserida precisa de um nó origem e nó destino.

Um nó ou uma aresta ao ser inserido no diagrama é selecionado automaticamente, o que significa que o elemento inserido passa a ser o foco das subsequentes operações de edição. Se o utilizador digitar e a *label* do elemento for editável, o texto na *label* do elemento será substituído pelo texto inserido, e se ele arrastar um dos *handlers* (os pequenos quadrados nas bordas), o nó será redimensionado, se a configuração do nó não for definida para redimensionamento automático. Nesse caso, o nó é ajustado automaticamente de acordo com o texto da *label*.

O Eshu foi desenvolvido de modo a poder exibir *feedback* visual ao utilizador. No *feedback* os nós ou arestas a serem modificados ou eliminados são apresentados com menor transparência. E um nó a ser inserido é apresentado no diagrama com um tamanho superior em relação aos outros nós.

As imagens dos nós são as imagens SVG definidas no DL^2. Estas imagens são utilizadas como imagem de fundo dos nós. As arestas são desenhadas de acordo com as posições dos nós aos quais se encontram ligados.

5.2.1 Janela de propriedades

Alinhado com as tendências dos principais editores de diagramas do mercado, o Eshu 2.0 passa a ter uma nova janela de edição das propriedades, que permite exibir e alterar as propriedades dos elementos na seleção atual, bem como a configuração de algumas

das propriedades da área de desenho (grelha e cor de fundo).

No Eshu 2.0, conforme é apresentado na Figura 5.4, no canto superior direito foi adicionado um botão que permite mostrar ou ocultar a janela de propriedades. Essa janela, uma vez aberta, é apresentada no lado direito do editor e permanece ativa, sem afetar a execução do diagrama, facilitando a manipulação dos elementos, pois permite editar de forma rápida as propriedades dos elementos que fazem parte do diagrama.

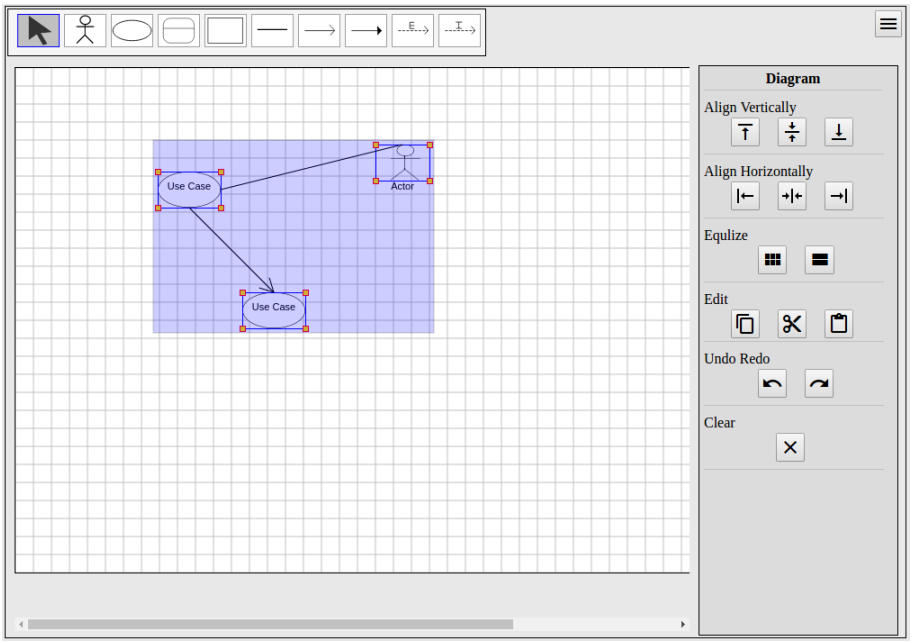


Figura 5.4: Janela de propriedades

A janela de propriedades exibe diferentes tipos de campos de edição, dependendo do estado do diagrama, isto é, somente os campos do editor ou elemento selecionados são visíveis. Esses campos de edição incluem *inputs* e botões para a edição das propriedades do editor ou dos elementos selecionados. As propriedades mostradas em cinza são somente de leitura. Na Figura 5.5 são apresentadas todas as *views* da janela. A janela `A` é apresentada quando não se encontra selecionado nenhum elemento no grafo e apresenta os campos para inserir ou ocultar a grelha, alterar a cor da linha da grelha e a distância entre as linhas da grelha. Além disso, permite alterar a cor de fundo da área de desenho de diagrama, colar elementos no diagrama, refazer uma ação no grafo ou apagar todos os elementos do diagrama. Na parte inferior da janela é apresentado um botão de ajuda que abre um *link* com informações sobre o editor Eshu. A janela `B` é apresentada quando uma aresta é selecionada, permite ver o tipo da aresta, editar a *label*, as cardinalidades se existirem, aumentar ou diminuir a sobreposição do elemento no diagrama e eliminar a aresta. A janela `C`

e `D` apresentam os campos para editar e configurar um nó selecionado no diagrama. Na janela `C` os campos apresentados são para um nó simples (caso de uso), onde é possível editar a *label*, a posição, a dimensão, o nível de sobreposição em relação aos outros elementos ou apagar o nó. A janela `D` é também para um nó, mas neste caso, o nó é complexo e contém mais do que uma *label*. No exemplo em concreto, trata-se de um nó classe. É adicionado um campo na janela para cada propriedade da classe (atributo ou operação) onde é possível editar, adicionar ou remover essas propriedades. Tanto na janela apresentada para o nó como para a aresta, na parte inferior da janela é apresentado um campo (botão) que redireciona o utilizador para uma página ou um vídeo com informações sobre o tipo de elemento selecionado. Os campos da janela `E` são apresentados ao serem selecionados múltiplos nós no editor (retângulo seleção ativo), e apresentam campos que permitem alinhar os elementos, tanto verticalmente como horizontalmente. Permite também equalizar os elementos selecionados ao tamanho do menor ou maior elemento selecionado, cortar, copiar, colar ou apagar os nós selecionados.

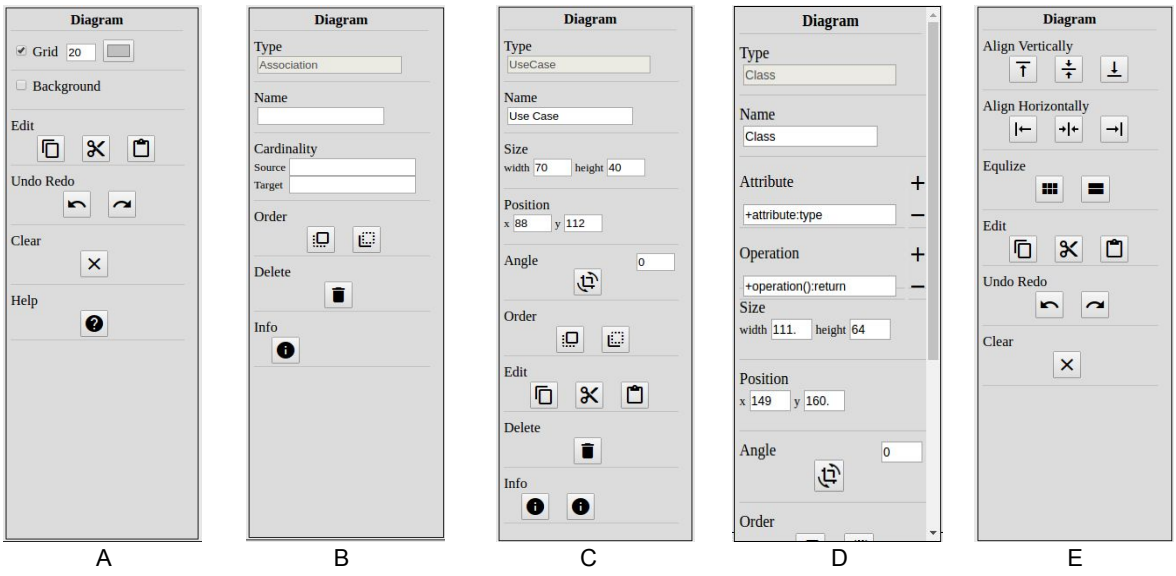


Figura 5.5: Views da janela de propriedades

5.3 Extensibilidade

Eshu foi projetado para ser extensível, de modo a poder incorporar novas linguagens diagramáticas. Uma das principais melhorias apresentadas pelo Eshu 2.0 é a parte de extensibilidade de nós e arestas. No Eshu 1.0, a criação de um novo tipo de nó (ou

aresta) envolve a criação de uma nova classe que estende uma outra classe `Vertice` (ou `Edge` para arestas) e é necessário definir o método `draw`. Na nova versão Eshu 2.0 criou-se a linguagem de configuração DL^2 onde é especificado os tipos de nós e arestas de uma determinada linguagem. Por isso, para criar um novo tipo de nó (ou aresta) é adicionada uma nova configuração do elemento `nodeType` (ou `edgeType`) no ficheiro de configuração. Este elemento contém informações gerais para um nó (ou aresta), como por exemplo o caminho de imagem SVG (usado para representa-lo na UI), nome do tipo, restrições nas conexões, entre outros, que estão especificados no capítulo 6. No Eshu 2.0 não é necessário definir o método *draw*, uma vez que é usado a imagem SVG adicionado no ficheiro configuração para representar esse elemento no editor de diagrama.

Foram adicionadas à API do Eshu novas funções de modo a suportar a criação do novos tipos de elementos (nós e arestas) a partir do DL^2. São elas `createNodeType`, `createEdgeType`, `createNodeTypes` e `createEdgeTypes`. O `createNodeType` e o `createEdgeType` recebe uma configuração específica para um nó ou aresta e cria um objeto `nodeType` (ou `edgeType`). Já o `createNodeTypes` e `createEdgeTypes` recebem uma *string* com as configurações dos tipos de nós e arestas do diagrama, converte a *string* para objeto *JSON* de modo a aceder as configurações definidas e cria os tipos de nós e arestas para o diagrama. Os tipos de nós e arestas criados são apresentados na `toolbar` como elementos a serem inseridos no diagrama.

5.4 Resumo

Neste capítulo foi apresentado a nova versão do editor de diagrama Eshu, nomeadamente a sua arquitetura e a implementação. Uma das principais diferenças na arquitetura e implementação do Eshu 2.0 consiste na separação da parte de dados (grafo) da parte da visualização e a introdução dos comandos *Undo* e *Redo*. Foi introduzida no Eshu uma nova janela de edição que permite exibir e alterar as propriedades de um elemento ou um grupo de elementos selecionados na área de desenho ou ainda alterar algumas propriedades do estilo do editor como a grelha e a cor de fundo.

Uma das grandes inovações do Eshu 2.0 foi na parte de extensibilidade, pois o editor de diagrama passou a ser configurado por ficheiro no formato DL^2. Este tipo de documento XML será descrito no capítulo seguinte. No Eshu 1.0, para estender o editor de diagramas a novos tipos de diagramas é necessário editar o código fonte para criar novos tipos de elementos. Enquanto que no Eshu 2.0 foi criado um tipo

de documento XML para definir os tipos de nós, arestas e restrições do tipo de diagrama . Este tipos de elementos são depois carregados para uma estrutura no Eshu, denominados `Nodetype` e `EdgeType`, consoante o caso que guardava todas as configurações gerais do elemento como a imagem, nome do tipo e restrições da ligações, entre outros.

Capítulo 6

Linguagem de configuração - DL^2

Tanto o Kora como o Eshu foram projetados para serem extensíveis a novos tipos de diagramas, isto é, ser fácil e prático incorporarem novos tipos de diagramas no editor. Para isso, foi definida uma linguagem de configuração, chamada DL^2 (*Diagrammatic Language Definition Language*), que consiste num ficheiro XML com sintaxe em XML Schema. Este ficheiro, além de incluir as configurações do estilo, do editor e da barra de ferramentas, inclui também as especificações de sintaxe da linguagem diagramática, ou seja, configurações sobre os tipos dos nós (tipo e número mínimo e máximo de ligações), arestas (tipo da aresta e tipo de nó origem e destino) e as restrições da linguagem (ex: os tipos de ligações permitida pela linguagem). O tipo de documento DL^2 foi definido utilizando XML Schema. O ficheiro instância é configurado no perfil de administração do Mooshak, e segue as especificações definidas no ficheiro XML Schema da linguagem DL^2.

6.1 Principais elementos do DL^2

A figura 6.1, apresenta um diagrama de classe com os principais tipos de elementos e seus atributos definidos no ficheiro XML Schema da linguagem de configuração DL^2.

6.1.1 Elemento DL2

O elemento principal (*root*) do ficheiro XML Schema da linguagem DL^2 é DL2, que é constituído por dois elementos: `style` e `diagram`. O elemento `style` é do tipo

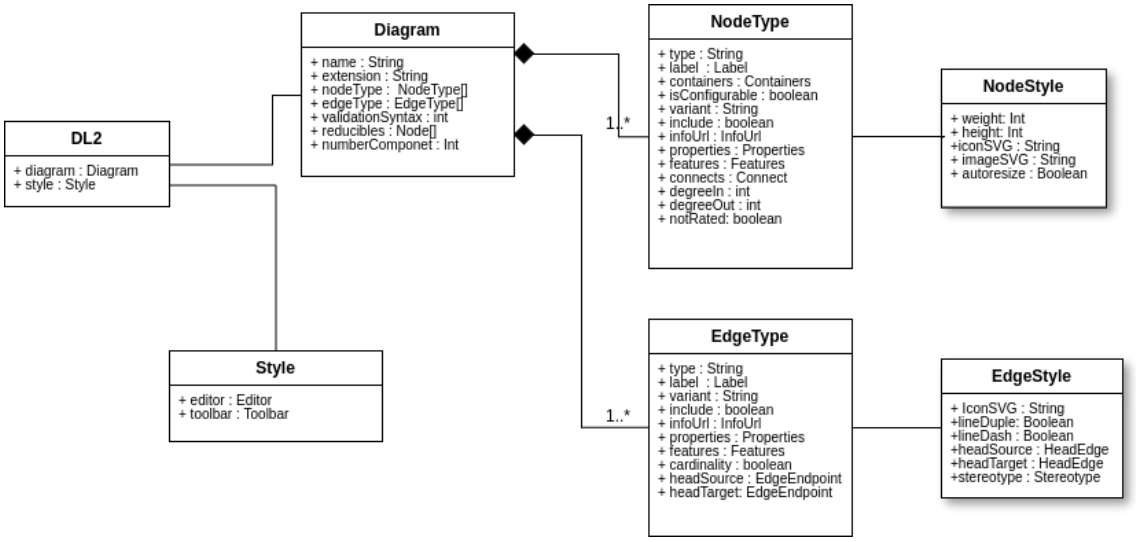


Figura 6.1: Diagrama Classe do DL2

`Style`, e contém os elementos que definem o estilo do editor (`editorStyle`) e da `toolbar` (`toolbarStyle`). O elemento `diagram` contém os elementos que definem a linguagem diagramática, isto é, elementos que definem os nós, as arestas e as restrições da linguagem a ser apresentada no editor de diagrama.

6.1.2 Elemento Diagram

No tipo `Diagram` encontram-se definidos atributos e elementos para a definição e configuração da sintaxe da linguagem. Este contém dois atributos, um que define o nome da linguagem e outro que define a extensão da linguagem. Além dos atributos, tem definido 2 elementos: `nodeTypes` e `edgeTypes`. Estes permitem configurar os tipos dos nós e arestas da linguagem. Contém ainda um elemento `syntaxeValidation` que possibilita a configuração das regras de sintaxe da linguagem.

NodeType

O tipo `NodeType` define os elementos responsáveis pela configuração dos tipos de nós que fazem parte do diagrama, tais como, o nome e o grupo do tipo do nó, as configurações da etiqueta, as `anchors`, os `handlers` e as propriedades. Permite ainda definir estilos para um determinado tipo de nó, como largura, altura, imagem SVG, imagem do ícone, tipo, rotação, entre outros. Além disso, permite definir as regras de sintaxe para a ligação ao nó, como o número de entrada e saída para cada tipo de aresta e nó do diagrama, ou tipo de nó e aresta a que se pode ligar. O `Nodetype` permite

definir se o tipo do nó é simples ou complexo. O tipo simples contém apenas uma *label* e a imagem do nó é a imagem definida pelo estilo, enquanto que o tipo complexo, além da *label* de editar o nome do nó, pode conter `Containers` (apêndice **??**), que são contentores dentro da *label* em que podemos adicionar, remover e editar etiquetas. A imagem do tipo complexo é um quadrado, mas também podemos configurar na parte do estilo para ser um círculo. O tipo complexo foi definido para tratar casos especiais, como a classe no diagrama de Classe. Através do elemento `includeElement` é possível definir ligação no grafo de um determinado nó para outro nó apenas pela sobreposição de nós, isto é, se um nó A sobrepõe um nó B, então é considerada uma ligação do nó A para o nó B. Outros aspetos importantes do `Nodetype` é que permite definir um conjunto de URLs com informações sobre o tipo do nó. Estes URLs são apresentados na janela de propriedades e, também, usados pelo avaliador no *feedback* deste tipo de nó. No apêndice B.3.1 é apresentado a estrutura do `NodeType`.

EdgeType

O tipo `EdgeType` é composto por elementos que configuram os tipos de arestas que fazem parte da linguagem. Estes elementos configuram a parte do estilo da aresta, como o nome do tipo e do grupo da aresta, o tipo de seta na origem e destino da aresta, o tipo de linha da aresta (simples, dupla ou a tracejado), a *label* da aresta, as propriedades que vão ser importadas e exportadas para Json e um *stereotype* para a aresta. Do mesmo modo que o tipo `Nodetype`, o tipo `EdgeType` contém um elemento que permite definir um conjunto de URLs a ser adicionado no *feedback* e apresentado na janela de propriedades como ajuda. No apêndice B.3.2 é apresentado uma descrição pormenorizada sobre o tipo `EdgeType`.

SyntaxeValidation

O tipo `SyntaxeValidation` é formado por atributos que definem e configuram a validação sintaxe do diagrama realizado pelo avaliador Kora. Este tipo contém atributos que permitem definir o número mínimo e máximo de componentes do diagrama, os tipos de validação de sintaxe a serem feitas (nos nós, arestas e/ou números de componentes) e onde é feita, no Eshu ou no avaliador de diagrama. Este tipo contém um elemento que permite definir os tipos de nós que se pode reduzir a propriedades na avaliação de diagramas, isto é, são tipos simples normalmente constituídos apenas por uma *label* e que na avaliação de grafos deixam de ser nós e passam a ser propriedades dos nós a que se encontram ligados. Como exemplo, no diagrama ER o tipo `Atributo` é indicado como tipo redutível e na avaliação de grafo, os nós do tipo `Atributo` são reduzidos a propriedades dos nós a que se encontram ligados (Entidade

ou Relacionamento). Esta redução permite diminuir o tamanho do grafo, e assim melhorar a performance do avaliador de grafo, uma vez que é mais fácil comparar propriedades do que tipos. No apêndice B.2 é apresentado mais detalhes sobre o tipo `SyntaxeValidation` .

6.1.3 Elemento Style

O elemento `style` define as regras que configuram o estilo do editor. Este é constituído por dois elementos: `editorStyle` e `toolbarStyle`. O elemento `editorStyle` é formado por atributos que definem a configuração do estilo do editor: a largura, a altura, o estilo do borda do editor e a cor de fundo na área de desenho. Este também permite mostrar uma grelha na área de desenho do editor e definir a cor da linha da grelha. No elemento `toolbarStyle` encontram-se definidos os atributos que definem a configuração do estilo da `toolbar` do Eshu, nomeadamente a largura, a altura, a borda, o tamanho dos ícones dos nós e arestas, a cor de fundo e a posição da janela. No apêndice B.2 é apresentado mais detalhes sobre este elemento.

6.2 Resumo

Neste capitulo foram abordados os principais tipos do componente DL^2, uma linguagem de definição de linguagens diagramáticas que foi introduzida no ambiente de avaliação de diagramas do Mooshak para permitir a sua extensibilidade. Esta linguagem de definição é um ficheiro XML com sintaxe em XML Schema, que inclui as configurações dos nós e arestas e a especificação sintática da linguagem diagramática, e ainda as configurações de estilo do editor de diagramas.

O componente DL^2 permite também configurar as regras para a avaliação sintática no Kora. Além disso, permite atribuir URLs com informações sobre o elemento que será apresentado no *feedback* e na janela de propriedades do elemento. O DL^2 contém elementos que permitem definir características particulares dos elementos, como por exemplo, o elemento `container` (para agrupar propriedades no nó) e `include` (para ligação entre nós por sobreposição) da definição do tipo de nó (`nodetype`). No apêndice B são apresentados em detalhe todos os elementos da linguagem configuração $DL2$.

Capítulo 7

Validação

O objetivo dos componentes apresentados nas secções anteriores é melhorar a avaliação do diagrama no Mooshak. Em particular, espera-se que os novos componentes possibilitem o suporte a qualquer tipo de diagrama, melhorem a qualidade dos *feedbacks* e resolvam os problemas de usabilidade. As secções a seguir apresentam as validações realizadas para avaliar se esses objetivos foram cumpridos.

7.1 Expressividade da linguagem de configuração

Um objetivo importante deste trabalho é permitir o suporte de novas linguagens diagramáticas. Para esse efeito, como é apresentado no capítulo anterior, foi desenvolvida uma nova norma XML para a especificação de linguagens diagramáticas, denominada DL^2. Para validar a expressividade da linguagem de configuração proposta, várias linguagens diagramáticas foram configuradas a partir dela.

A linguagem DL^2 define as configurações sintáticas de uma linguagem diagramática, que é usada na configuração do editor de diagramas, na conversão de diagramas para a representação de grafo e na geração de *feedback*.

O Mooshak já suporta o conceito de configuração para avaliação de programas. Contudo, as configurações disponíveis foram projetadas para linguagens de programação. Estas incluem algumas funcionalidades como linhas de comando de compilação e execução para cada linguagem. Para suportar a configuração de linguagens diagramáticas, foi adicionado na janela de configuração um campo com a opção de adicionar um arquivo de configuração. Na configuração para linguagem diagramática

é adicionado as especificações do DL^2 nesse campo.

A versão anterior suportava apenas diagramas EER. Por isso, esse tipo de diagrama foi o primeiro candidato a testar a expressividade do DL^2. Tem doze tipos de nós e três tipos de arestas, e nenhum deles colocou qualquer dificuldade. Em particular, todos os tipos de nó foram fáceis de desenhar no SVG e estes tipos de nó e arestas possuem um pequeno e simples conjunto de propriedades. Após desenhar os SVG dos nós e ícones para nós e arestas, estes foram adicionados a um arquivo ZIP junto com um XML com configuração de elementos.

O UML é uma linguagem de modelagem visual com vários tipos de diagramas que são amplamente utilizados em ciência da computação. Para validar a abordagem proposta, selecionamos diagramas de classe e de casos de uso, pois estes são frequentemente usados em cursos que abrangem UML. Cada uma dessas duas linguagens exigem características particulares de DL^2 na sua definição. O diagrama de casos de uso define relações entre nós sem usar bordas: o sistema é representado como um retângulo contendo casos de uso. O elemento `include` de DL^2 permite a definição de conexões entre nós sobrepostos e foi usado para criar essas relações implícitas. As classes no diagrama de classe também são particularmente desafiadoras, pois possuem propriedades complexas, como as que representam atributos e operações. No elemento `container` da DL^2, as definições provaram a sua utilidade na estruturação dessas listas de atributos e complexos. Na figura 7.1 são apresentados alguns exemplos de nó configurados a partir de DL^2 e na apêndice C é apresentado exemplos de configurações para o nó classe (diagrama de classe) e aresta `extend` (diagrama de caso de uso).

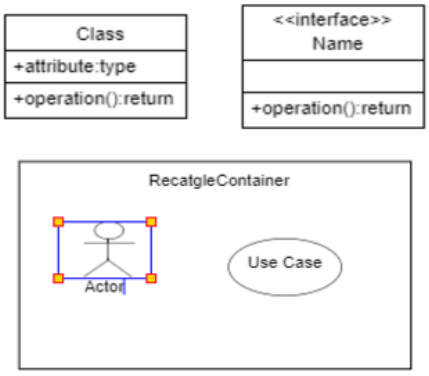


Figura 7.1: Exemplo de Nós

7.2 Métricas de avaliações

O aprimoramento de *feedback* também é um dos principais objetivos desta investigação e seria interessante se as mudanças introduzidas resultassem em uma variação significativa nas métricas de avaliação, como o tempo para resolver um exercício depois de receber o primeiro *feedback*, deve diminuir. Um experimento foi projetado para testar a hipótese de que essas métricas mudaram com a introdução do *feedback* aprimorado. Os participantes foram alunos de licenciatura do departamento de informática. O número de participantes foi de 27, dos quais 7 eram do sexo feminino, e a média de idade era de 20,83 anos. Eles tentaram resolver um conjunto de 3 exercícios ER.

Dois grupos foram criados para analisar o impacto de Kora no melhoramento do *feedback*: um grupo de controle, que usou a versão anterior; e um grupo de tratamento, que usou a nova versão. Esses dois grupos foram criados a partir de pares de participantes e exercícios, em vez de apenas de participantes. Todos os participantes tiveram experiência de resolver exercícios com e sem Kora, pelo que poderiam ser solicitados a comparar ambos os tipos de *feedback*, como explicado na secção a seguir.

Um novo modo de avaliação foi implementado no Mooshak para suportar esta configuração experimental. O Mooshak escolhe ou o Kora ou o modo de avaliação de diagrama anterior, com base em uma função *hash* calculada a partir do `ID` do participante e do `ID`. Esta função garante que exercícios sucessivos usem modos diferentes. Portanto, se um determinado aluno receber *feedback* filtrado por Kora no primeiro exercício, ela recebe comentários não filtrados no segundo exercício e novamente filtrou o *feedback* no terceiro. A função `hash` garante também que os participantes com *IDs* consecutivos tenham um modo diferente para o primeiro exercício. Os 27 alunos que resolveram cada 3 exercícios produziram 447 envios que foram agrupados pelo uso de Kora. Os grupos que usaram Kora tiveram 27 pares de exercícios estudantis diferentes e o grupo de controle teve 29.

Infelizmente, os resultados obtidos nesta experiência não foram conclusivos. Tanto com ou sem o uso de Kora, a média de todas as métricas foi semelhante. Em particular, o número médio de tentativas com Kora é ligeiramente maior, o que pode sugerir que os alunos não conseguiram facilmente deduzir a solução do conjunto completo de diferenças. O que podemos considerar positivo, pois o *feedback* não gera demasiada informação ao aluno. No entanto, a hipótese de que esses dois grupos serem diferentes não é validada pelo teste *t de Student*. Serão necessários mais testes, com mais participantes e exercícios de diferentes níveis de dificuldade para discriminar melhor

o impacto de Kora em diferentes situações.

7.3 Usabilidade e Satisfação

A experiência para avaliar a usabilidade e satisfação da versão anterior consistiu no uso do sistema nas aulas práticas da unidade curricular de Bases de Dados, no **Departamento de Ciência de Computadores da Faculdade de Ciências da Universidade do Porto (FCUP)**. Após a experiência, os alunos foram convidados a preencher um questionário *online* (apêndice E) baseado no modelo Nielsen [29], no *Google Forms*. As respostas revelaram deficiências de velocidade, confiabilidade e flexibilidade. Os estudantes queixaram-se principalmente de dificuldades na construção de diagramas e o alto atraso ao avaliar seus diagramas.

Para verificar o impacto das mudanças, a validação da usabilidade da versão atual seguiu uma abordagem similar. A experiência ocorreu nos dias 16 e 19 de junho de 2017, também com alunos de graduação matriculados no mesmo curso.

O questionário era muito semelhante ao usado anteriormente, mas desta vez, foi incorporado no Enki, como um recurso do curso. Além disso, o novo questionário inclui um grupo de perguntas especificamente sobre *feedback*, para avaliar se a Kora ajuda os alunos no seu caminho de aprendizagem, sem fornecer a solução diretamente.

A Figura 7.2 mostra os resultados agrupados pelas heurísticas da Nielsen das versões anterior e nova. Os dados coletados são mostrados em dois gráficos de barras, sendo as heurísticas ordenadas por ordem decrescente de satisfação do usuário.

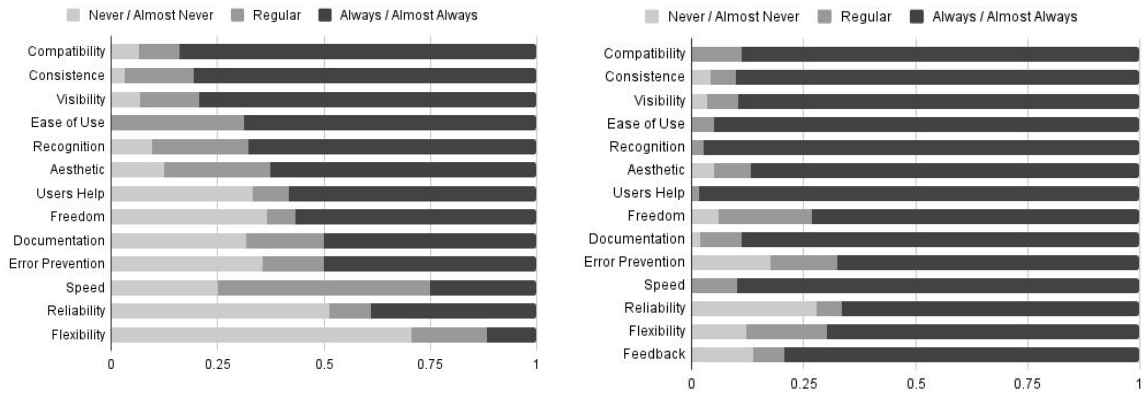


Figura 7.2: Avaliação de aceitabilidade - no lado direito os resultados da versão anterior, e no lado direito os resultados da nova versão

É claro que a usabilidade do sistema e a satisfação dos utilizadores melhoraram. Na verdade, todas as heurísticas obtiveram melhores resultados. Além disso, os resultados mostram que, com a nova versão, as heurísticas com maior satisfação são a ajuda, o reconhecimento e a facilidade de uso pelos utilizadores. Por outro lado, a confiabilidade, a prevenção de erros e a flexibilidade foram as áreas com piores resultados. Alguns alunos reclamaram que o *feedback* com Kora em alguns caso é muito explícito, o que possivelmente lhes permite chegar a uma solução por várias tentativas e erros.

A última questão do questionário é uma classificação geral do sistema em uma escala de tipo *Likert* de 5 valores (muito bom, bom, adequado, mau, muito mau). A maioria dos alunos (57.1%) classificou-o como muito bom, enquanto o resto (42.9%) declarou que era bom.

Capítulo 8

Conclusão

Este capítulo apresenta um resumo do trabalho que esteve na base desta dissertação, destacando as principais contribuições que dele resultaram. São também indicados alguns tópicos para a continuação do desenvolvimento do ambiente de aprendizagem de diagramas do Mooshak 2.0.

8.1 Trabalho Desenvolvido

Mooshak é um sistema que suporta avaliação automática em ciência de computadores e tem sido usado tanto para programação competitiva como para *e-learning*. Recentemente, foi complementado com a avaliação dos diagramas *Entity-Relationship* (ER) e *Extended ER* (EER). Os diagramas nessas linguagens são criados com um editor de diagramas e convertidos em grafos. Os grafos dos diagramas de alunos são avaliados comparando-os com grafos obtidos a partir dos diagramas de solução. A experiência adquirida com esta ferramenta revelou uma série de deficiências que foram colmatados no presente trabalho.

Uma das principais contribuições deste trabalho é a linguagem DL^2. Os documentos XML que utilizam esta linguagem de configuração desacoplam as definições sintáticas do código-fonte e simplificam o suporte a novos tipos de linguagens diagramáticas. As configurações no DL^2 são usadas tanto no cliente como no servidor. Do lado do cliente, eles são usados para o editor de diagramas Eshu, para configurar a GUI com a sintaxe visual dos tipos de nó e aresta das linguagens. Do lado do servidor, eles são usados pelos componentes Kora para realizar análises sintáticas, como pré-

requisito para a análise semântica. Essas configurações também são instrumentais na integração com conteúdo estático que descreve a sintaxe da linguagem, que pode ser usado como *feedback* quando são detetados erros. A expressividade de DL^2 foi validada pela reimplementação de editores de diagramas UML, bem como ER e EER.

Outra contribuição deste trabalho são as abordagens utilizadas pelo componente Kora do lado do servidor. Em complemento às contribuições relativas à sintaxe do diagrama que conduziram DL^2, mencionadas no parágrafo anterior, a sumarização das mensagens de *feedback* também contribui para melhorar a qualidade do *feedback*. O comparador gráfico usado para análise semântica produz uma grande quantidade de erros, que confundem os alunos tanto quanto ajudam. O resumo proposto consegue gerar mensagens concisas e relevantes, começando com mensagens gerais agregando vários erros e avançando para erros mais focados e particulares se a dificuldade do aluno persistir. No último caso, o *feedback* é gerado na janela de edição do diagrama usando a sintaxe visual da linguagem diagramática.

O redesenho e a reimplementação do editor Eshu contribui para colmatar as lacunas apresentadas na validação anterior a nível de usabilidade do ambiente de avaliação de diagramas no Enki e, sobretudo, melhorar a parte de extensibilidade a novas linguagens diagramáticas.

A longo da elaboração deste trabalho foram publicados e apresentados dois artigos científicos. Um artigo titulado *Enhancing feedback to students in automated diagram assessment* [7], apresentado em Vila do Conde durante o SLATE'2017, e um segundo com título de *Improving Diagram Assessment in Mooshak*, conferência TEA em Barcelona (Espanha). O Mooshak com Kora está disponível para *download* na página inicial do projeto Mooshak e também encontra-se disponível uma instalação do Mooshak configurada com alguns exercícios de diagrama ER (em inglês) para teste *online* [1].

8.2 Trabalho Futuro

O objetivo principal desta dissertação era desenvolver um ambiente de aprendizagem da linguagem UML, com base na resolução de exercícios práticos. Este objetivo foi alcançado, contudo algumas ideias e sugestões de melhorias surgiram ao longo do desenvolvimento do nosso trabalho. De seguida, são apresentados tópicos para futuras pesquisas:

[1] http://mooshak2.dcc.fc.up.pt/Kora

- ***Feedback* para encorajar e motivar o aluno na resolução de exercícios** – Ter um mecanismo no Kora que congratula o aluno sempre que este consiga resolver um exercício ou aumentar relativamente a sua classificação após várias tentativas (técnicas de gamificação).

- **Técnicas para evitar o abuso de recursos para obter feedback mais direto** – Fazer com que o *feedback* seja uma fonte de ajuda e não um sistema que oferece parte da solução. O sistema Kora, por apresentar *feedback* progressivo, leva a que o aluno submeta várias vezes o mesmo diagrama de modo a que o Kora incremente os detalhes no *feedback*. Por isso, será interessante ter algum mecanismo, como por exemplo, a limitação no número de submissões ou no tempo entre as submissões.

- **Melhorar o estilo do editor de diagrama Kora** – Melhorar os estilos e os botões do editor Kora cliente no Enki, para aderir ao *Material Design* que está atualmente a ser implementado no Enki.

- ***Feedback* em mais linguagens** – Neste momento, o *feedback* textual é apresentado em português ou inglês. Uma das melhorias seria adicionar mais linguagens, como espanhol e francês ao *feedback* textual.

- **Melhorar o *feedback* visual** – Atualmente, no caso do *feedback* visual ser para modificar ou remover um elemento, é apresentado no diagrama destacando o elemento através da redução da transparência. No caso do *feedback* ser para inserir um nó, é adicionado o nó com tamanho aumentado na área de desenho. Para o melhorar, pode-se adicionar uma cor diferente ao *feedback* visual, consoante o tipo de modificação sugerida pelo avaliador ao grafo (inserir, eliminar, editar). Para isso, será necessário editar a imagem SVG do nó.

- **Facilitar a definição de linguagens** – Atualmente o Mooshak tem um editor XML interativo que poderia ser usado na autoria de ficheiros de configuração em DL2, das linguagens diagramáticas usadas no Kora. Para tanto, será então necessário criar um JSON schema com as regras de validação da DL2 para o usar.

Apêndice A

Estudo comparativo sobre os editores de diagrama

A.1 Suporte de diagramas

Na tabela A.1 é apresentado um conjunto de ferramentas de edição de diagramas UML e os tipo de diagramas UML suportada por estas ferramentas.

A.2 Características de alguns dos editores de diagrama

A tabela A.2 apresenta um estudo comparativo entre as ferramentas UML, comerciais, livres e web.

Tabela A.1: Ferramentas UNL e os tipos de diagramas que se pode modelar

		Diagrama De													
		Atividade	Caso de uso	Classe	Colaboração	Comunicação	Componentes	ER	Estruturas Compostas	Instalação	Máquina de Estado	Objetos	Pacotes	Perfis	Sequência
Livre	ArgoUML	✓	✓	✓	✓					✓	✓				✓
	Dia	✓	✓	✓	✓		✓	✓	✓	✓	✓	✓	✓		✓
	PlantUML	✓	✓	✓	✓						✓	✓			✓
	Umbrello	✓	✓	✓	✓			✓	✓	✓	✓	✓	✓		✓
	UML Designer	✓	✓	✓	✓						✓	✓	✓	✓	✓
Comercias	MagicDraw	✓	✓	✓	✓	✓		✓	✓	✓	✓	✓	✓	✓	✓
	Modelio	✓	✓	✓	✓			✓	✓	✓	✓	✓	✓		✓
	StarUML	✓	✓	✓	✓	✓				✓		✓	✓	✓	✓
	Visio	✓	✓	✓	✓	✓	✓	✓				✓			✓
	Rational Rose		✓	✓	✓		✓				✓	✓			✓
	UModel	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	
Web	Cacoo	✓	✓	✓	✓						✓		✓		✓
	Creately	✓	✓	✓	✓	✓	✓	✓			✓	✓	✓		✓
	Gliffy	✓		✓											
	Lucidchart	✓	✓	✓	✓						✓				

Tabela A.2: Comparação entre os tipo de ferramentas UML

		Características								
		Sistema de Validação	Exemplo e tutorial	Template	Janela Propriedades	Utilizar elementos de diferentes diagramas	Editar Elemento diretamente	Chat	É necessário ter uma conta	Editar diagramas em simultâneo
livres	ArgoUML	✓			✓		✓			
	Dia					✓	✓	✓		
	PlantUML						✓			
	Umbrello	✓			✓		✓			
	UML Designer	✓	✓	✓	✓	✓	✓			
	MagicDraw	✓	✓	✓	✓		✓			
	Modelio	✓	✓	✓	✓		✓			
comerciais	StarUML	✓				✓	✓			
	Visio	✓				✓	✓			
	Rational Rose				✓		✓			
	UModel	✓			✓	✓	✓			
web	Cacoo		✓	✓	✓	✓	✓	✓	✓	✓
	Creately			✓		✓	✓	✓	✓	✓
	Gliffy						✓		✓	✓
	Lucidchart	✓					✓		✓	✓

Apêndice B

API da linguagem de configurações – DL^2

B.1 DL^2

O elemento principal (root) do ficheiro DL2 XML Schema é DL2 que é constituído por dois elementos: o Style e o Diagram.

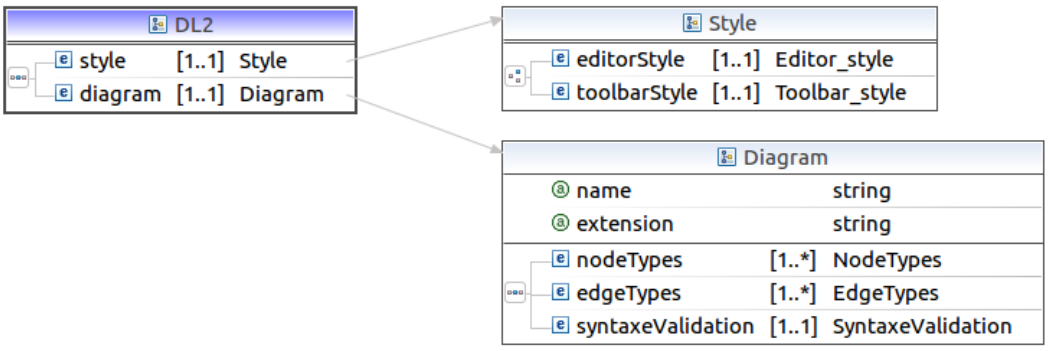


Figura B.1: Elemento DL2, style e Diagram

- O elemento style é do tipo Style, responsável por definir regras para a parte do estilo, contém os elementos que definem o estilo do editor (editorStyle) e da toolbar (toolbarStyle).
- O elemento diagram contém os elementos que definem a linguagem diagramática, isto é, elementos que definem os nós, arestas e as restrições da linguagem a serem apresentadas no editor de diagramas.

B.2 Style

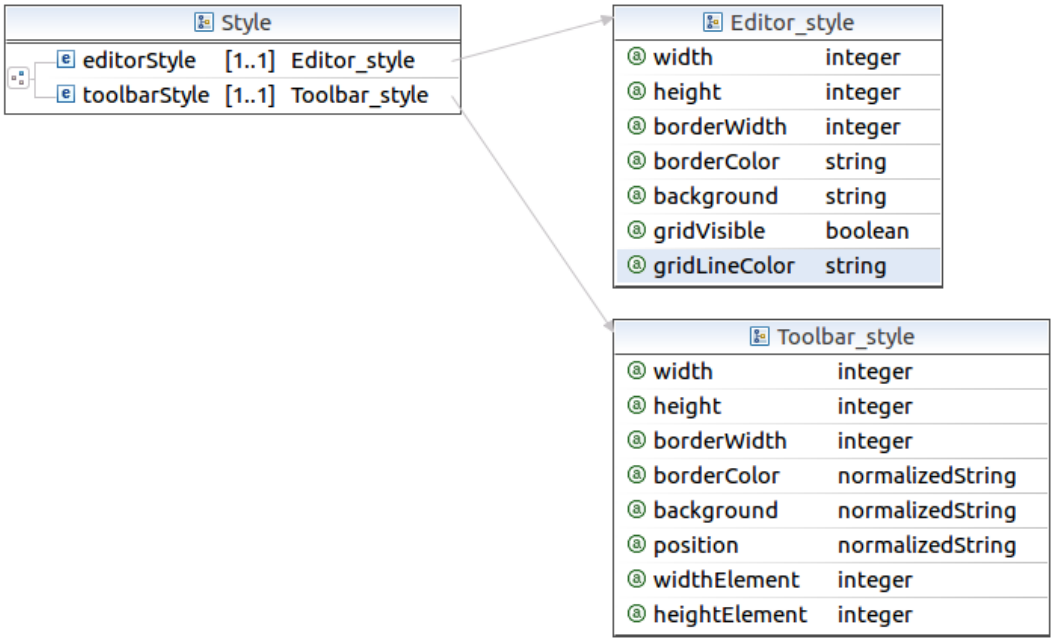


Figura B.2: Elemento Style, EditorStyle e Toolbar

O elemento Style, como o próprio nome indica, define as regras dos conteúdos que configuram o estilo do editor. É definido por dois elementos: o editorStyle e o toolbarStyle.

O elemento editorStyle é do tipo EditorStyle e este tipo é formado por atributos que definem a configuração do estilo do editor:

- width - a largura do editor
- height - altura do editor
- borderWidth - o tamanho da borda do editor
- borderColor - a cor da borda do editor
- background - cor de fundo da área de desenho.
- gridVisible - permite definir uma grelha na área de desenho do editor
- gridLineColor - define a cor da linha da grelha.

O elemento toolbarStyle é do tipo Toolbarstyle. No tipo Toolbarstyle encontram-se os atributos que definem a configuração do estilo da janela toolbar do Eshu,

- width - a largura da toolbar,
- height - a altura da toolbar
- borderWidth - o tamanho da borda do editor
- borderColor - a cor da borda
- widthElement - a largura dos ícones (nodetypes, edgetypes) na toolbar
- heightElement - a altura dos ícones (nodetypes, edgetypes) na toolbar
- background - a cor de fundo da toolbar
- position - a posição da janela (vertical ou horizontal)

B.3 Diagram

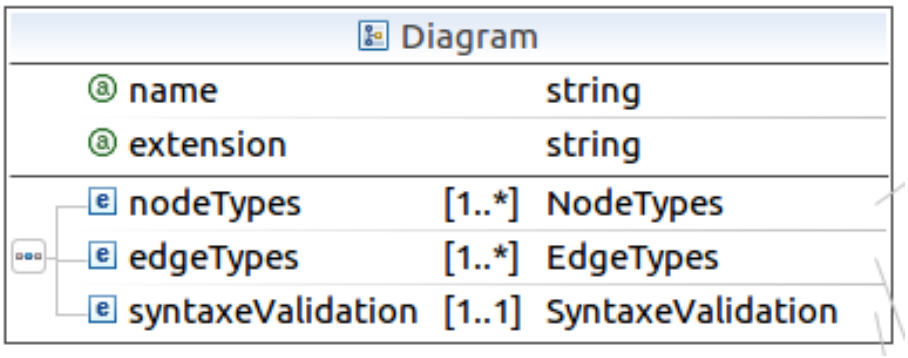


Figura B.3: Elemento Diagram

No tipo Diagram encontram-se definidos atributos e elementos para a definição e configuração da sintaxe da linguagem. Contém dois atributos:

- name - define o nome da linguagem
- extension - define a extensão da linguagem.

Além dos atributos, tem definido 3 elementos :

- nodeTypes - permite configurar os tipos dos nós da linguagem
- edgeTypes - permite configurar os tipos das arestas da linguagem
- syntaxeValidation - possibilita a configuração da regra sintaxe da linguagem.

B.3.1 Nodetype

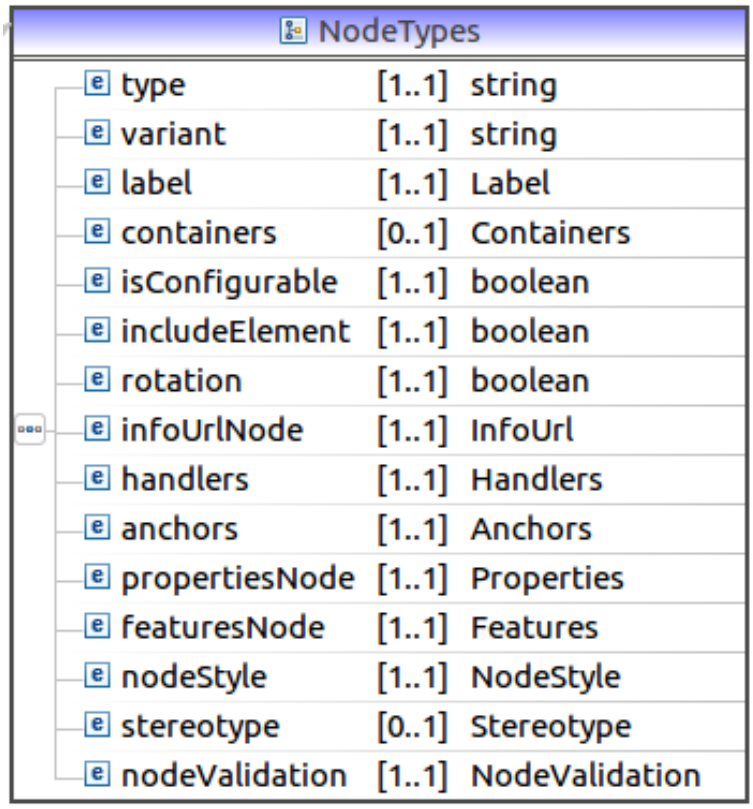


Figura B.4: Elemento Diagram

O tipo NodeType define os elementos que configuram os tipos de nós que fazem parte da linguagem. Estes elementos configuram o estilo do nó e as regras sintaxe do nó, são elas:

- `type` - define o nome do tipo de nó
- `variant` - define o grupo do nó ou aresta, isto é, os elementos que pertencem ao mesmo grupo são apresentados na mesma div na toolbar. É apresentado na toolbar o último selecionado e ocultado os outros. Ao passar o rato sobre o elemento que está a ser exibido é apresentado os outros tipos dos elementos que

pertencem a este grupo. Os elementos do mesmo grupo permitem a troca entre si de tipo no editor.

- `label` - configura a label do nó, nomeadamente o estilo, a posição nó, o texto default entre outras opções (ver tipo label).

- `isConfigurable` - define se o nó é simples ou complexo, no caso de ser simples contém apenas uma label e a imagem do nó é a definida no estilo. Caso contrário, é possível definir vários containers no nó onde é possível editar texto nas labels e também inserir ou eliminar as labels. O tipo de nó complexo por padrão tem a forma retangular, mas pode ser alterado para a forma circular conforme a configuração.

- `containers` (opcional)- configurações para estrutura dentro do nó que agrupa várias labes. Permite adicionar e remover labels dentro do mesmo grupo. Como exemplo de container, temos o atributo no diagrama de classe onde é definido um container, de nome atributo, e é possível editar o nome do atributo, adicionar ou remover os atributos e definir expressões regulares para validar os atributos inseridos.

- `includeElement` - Permite definir ligações entre nós por sobreposição, isto é, se um nó A sobrepor um nó B e o nó B tiver a opção includeElement como true é considerado uma ligação do nó A para o nó B.

- `rotation` - Define se é possivel rodar ou não o nó.

- `infoUrlNode` - Permite definir um conjunto de url com informações sobre o nó.

- `url` são apresentados na janela de propriedades e também usados pelo avaliador no feedback.

- `handlers` - Define os pontos no nó onde é possível selecionar para redimensionar. (ver tipo Hanlder)

- `anchors` - Define os pontos no nó a que se pode ligar a uma aresta. (ver tipo Anhors)

- `proprietesNode` - Define as configurações para as propriedades do nó, como por exemplo, quais são visíveis na janela de propriedades e quais são importados e exportados para o JSON, que pode vir a ser utilizado para a avaliação do diagrama. (ver tipo Property)

- `featuresNode` - Define um conjunto de propriedades que podem ser acrescentados ao nó. (ver tipo Feature)
- `nodeStyle` - Define a configuração do estilo do nó, nomeadamente a imagem do nó, imagem do ícone a ser apresentada na toolbar, a altura e largura default, e o tipo de redimensionamento, se é automático ou não.
- `nodeValidation` - define as regras sintaxe do nó, isto é, quais dos nós é que se podem ligar aos nós deste tipo e com quais tipos de arestas. Também define o número mínimo e número máximo de ligações para cada tipo de aresta e nós . (ver Connect)
- `stereotype` (opcional) - define e configura um stereotype para o nó.

B.3.2 Edgetype

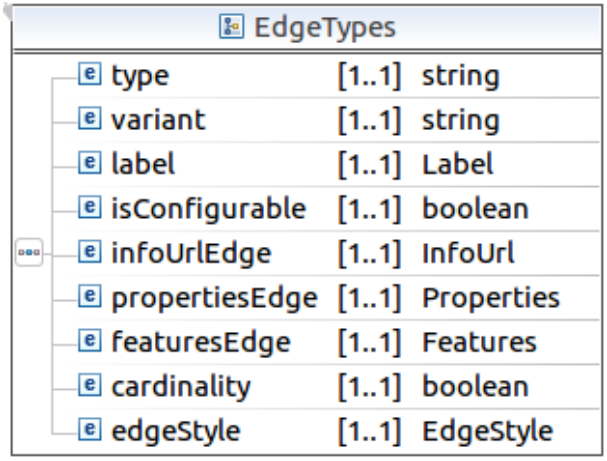


Figura B.5: Elemento Edgetype

O tipo EdgeType define os elementos que configuram os tipos das arestas que fazem parte da linguagem. Estes elementos configuram o estilo das arestas e as suas regras sintaxe:

- `type` - Define o nome tipo da aresta.
- `variant` - Tem a mesma função para variant do Nodetype. Define o grupo da aresta, isto é, os elementos que pertencem ao mesmo grupo são apresentados na mesma div na toolbar.

- `label` - Configura a label da aresta, nomeadamente o estilo, a posição nó, o texto default, entre outras opções. A label é apresentada no centro da aresta (ver tipo label).
- `infoUrlNode` - Permite definir um conjunto de url com informações sobre o tipo da aresta. Estes url são apresentados na janela de propriedades e também usados pelo avaliador no feedback.
- `proprietesNode` - Array de Property que define as configurações para as propriedades das arestas, como por exemplo, quais são visíveis na janela de propriedades e quais são importados e exportados para o JSON, que podem vir a ser utilizados para a avaliação do diagrama. (ver tipo Property)
- `featuresNode` - Define um conjunto de propriedades que podem ser acrescentados à aresta. (ver tipo Feature)
- `cardinality` - Define a configuração da cardinalidade para a aresta.
- `nodeStyle` - Define a configuração do estilo da aresta, nomeadamente a imagem do ícone a ser apresentada na toolbar, o tipo da seta na origem e destino da aresta. (ver HeadSource e HeadTarget)

B.3.3 SyntaxeValidation

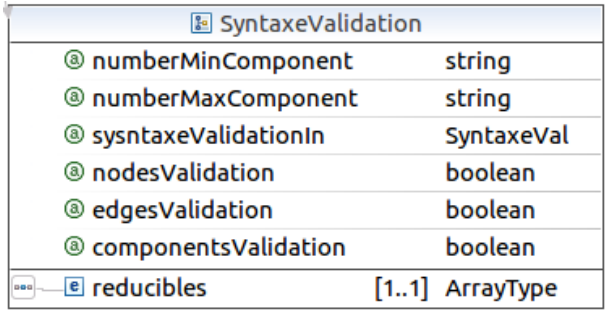


Figura B.6: Elemento SyntaxeValidation

O tipo SyntaxeValidation é formado por atributos que definem e configuram a validação sintaxe do diagrama realizado pelo avaliador Kora.

- `numberMinComponent` - número mínimo de componentes que o grafo que representa o diagrama deve ter.

- `numberMaxComponent` - número máximo de componentes que o grafo que representa o diagrama deve ter.
- `sintaxeValidation` - Permite definir onde é realizada a validação sintaxe, se diretamente no editor ou se é Kora.
- `nodesValidation` - Permite definir se é feita a validação sintaxe dos nós
- `nodesValidation` - Permite definir se é feita a validação sintaxe das arestas.
- `componentsValidation` - Permite definir se é feita a validação dos componentes na validação sintaxe ou não.
- `reducibles` - Define os tipos de nós simples da linguagem que são reduzíveis a propriedades na avaliação de diagrama pelo GraphEval

B.4 Tipos Auxiliares

B.4.1 Label

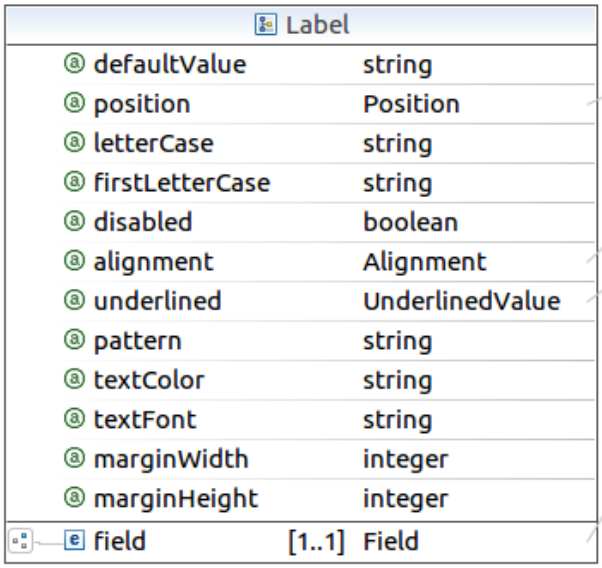


Figura B.7: Elemento Label

O tipo label define as configurações para a label dos nós e arestas. Labels no Eshu são criados a partir do tipo TextBox. Sendo assim, o tipo label configura o estilo, posicionamento e o texto default dos TextBox.

- `defaultValue` - Valor default do texto da lebal.
- `position` - A posição da label no nó e na aresta é sempre no centro.
- `letterCase` - Permite apresentar o texto da label em minúsculas.
- `firstLetterCase` - Permite apresentar em minúscula a primeira letra do texto da label.
- `disable` - Define se o texto da label é editável ou não, isto é, se o atributo tiver o valor true, a label é apresentada com o seu valor default, mas não é possível editá-lo.
- `alignment` - Define o alinhamento do texto na label. Pode ter valor (left, center e right)
- `underlined`- Define se o texto é sublinhado ou não. No caso do valor ser 0, não é sublinhado, valor igual a 1 é sublinhado e valor igual a 2 é sublinhado a tracejado.
- `pattern` - Define uma expressão regular para a validação do texto da label do nó.
- `textColor` - Define a cor do texto da label.
- `textFont` - Define a fonte do texto da label.
- `margemWidth` - Define a margem à esquerda e à direita da label no nó.
- `margemHeight` - Define a margem acima e abaixo da label no nó.
- `filed` - Permite indicar as partes do texto da label que serão usadas pela expressão regular e define em pattern para a validação sintaxe da label.

B.4.2 Container

O tipo container define uma parte do nó complexo em que podemos adicionar e remover labels. Como exemplo de container, temos o atributo no diagrama de classe onde é definido um container de nome atributo e é possível adicionar ou remover atributos e esses atributos.

- `name` - O nome do container.

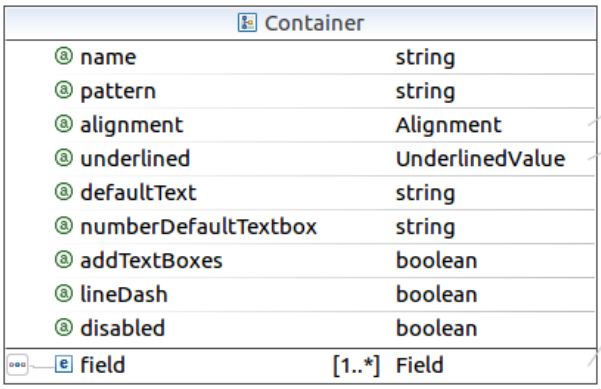


Figura B.8: Elemento Container

- `pattern` - Define uma expressão regular para a validação sintaxe das labels que pertencem a este container.
- `alignment` - Define o alinhamento das labels do container.
- `defaultText` - O texto default para as labels a serem criadas.
- `numberDefaultTextBox` - Número de textBoxs que aparecem no container ao ser criado um novo nó deste tipo.
- `addTextBoxe` - Permite definir se é possível adicionar ou não novos textBoxs no containers.
- `lineDash` - Define a linha superior do container com a tracejado.
- `disable` - Marca o container como desativada, isto é, não permite adicionar, remover ou editar label da container.
- `field` - Permite obter padrão a partir do pattern e é usada na validação sintaxe das labels do container.

B.4.3 Property

O tipo Property configura a visibilidade das propriedades na janela de prioridade utilizada para editar os nós e também na importação e exportação do elemento no formato JSON.

- `name` - Nome da propriedade.

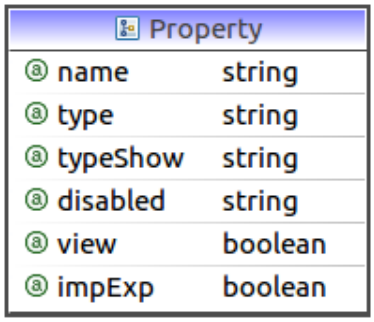


Figura B.9: Elemento Property

- `type` - O tipo da propriedade.
- `typeShow` - O tipo do elemento html à qual vai ser apresentado o valor da propriedade (input, button).
- `disabled` - No caso de ser true não é possível editar a propriedade.
- `view` - Se a propriedade é apresentado ou não.
- `impExp` - Se a propriedade é apresentado no Json do elemento.

B.4.4 NodeStyle

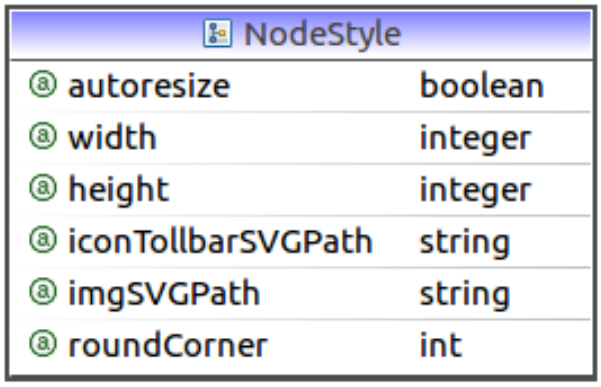


Figura B.10: Elemento NodeStyle

O tipo NodeTyle define as configurações do estilo para o tipo do nó.

- `autoresize` - No caso desse atributo ser verdadeiro é feito o alinhamento automático ao nó de acordo com o texto da label.

- `width` - A largura padrão do nó.
- `height` - A altura padrão do nó.
- `iconTollbarSVGPath` - O caminho do ficheiro SVG da imagem do ícone do nó.
- `imgSVGPath` - O caminho do ficheiro SVG da imagem do nó.
- `roundCorner` - (nó complexo) permite definir um valor utilizado para arrendodar os cantos do nó.

B.4.5 EdgeStyle

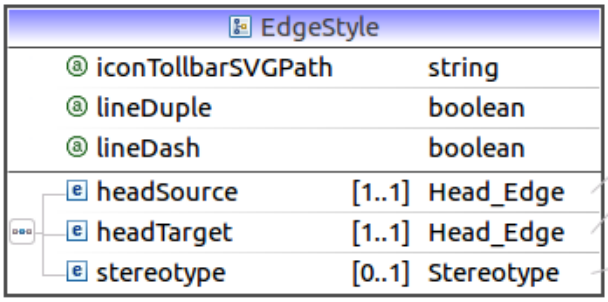


Figura B.11: Elemento EdgeStyle

O tipo EdgeStyle define as configurações para o estilo dos tipos da aresta.

- `inconTollbarSVGPath` - O caminho do ficheiro SVG da imagem do ícone da aresta.
- `lineDuple` - Permite definir uma linha dupla na aresta.
- `lineDash` - Permite definir a linha da aresta a tracejado.
- `headSource` - O tipo de seta na origem da aresta. O eshu já tem configurado vários tipos de setas: arrow-complete(uma seta simples), rhombus (losango), arrow-open (seta aberta na parte de trás) e sime-arrown (semi- seta). O valor none não apresenta qualquer seta na aresta.
- `headTarget` - O tipo de seta na origem da aresta.
- `stereotype` - Permite configurar e definir estilo para um stereotype da aresta.

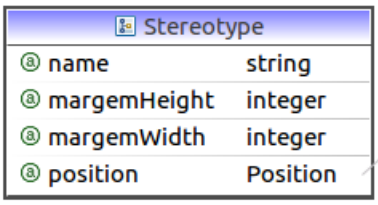


Figura B.12: Elemento Stereotype

B.4.6 Stereotype

- `name` - Nome do stereotype.
- `margemWidth` - Define a margem à esquerda e à direita do stereotype no elemento.
- `margemHeight` - Define a margem acima e abaixo stereotype no elemento.
- `position` - Define a posição do stereotype no elemento.

B.4.7 Connect

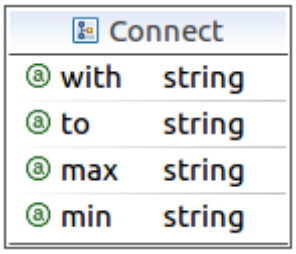


Figura B.13: Elemento Connect

Tipo que permite definir regras sintaxe para as ligações num nó que serão utilizados para a validação sintaxe.

- `with` - O nome do tipo da aresta.
- `to` - O nome do tipo do nó da outra extremidade da aresta.
- `max` - O número máximo de ligações entre tipos de nó definidos no ”to”com o tipo de aresta definidos em ”with”.
- `min` - o número mínimo de ligações entre tipos de nó definido no ”to”com o tipo de aresta definidos em ”with”.

B.4.8 Anchor

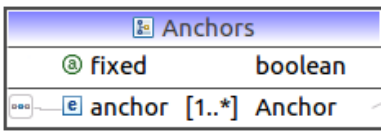


Figura B.14: Elemento Anchor

Este tipo define os pontos do nó em que é possível ligar a outros nós. No editor, por padrão, quando se cria uma aresta, a aresta é ligado ao anchor no nó mais próximo do outro nó. No caso do atributo fixed ter valor true a ligação passa a ser fixo ao anchor ligado.

Apêndice C

Exemplo de definição de tipo de nó e aresta no DL^2

C.1 Exemplo de configuração do nodeType Classe

A seguir é apresentado uma configuração para o nó classe de diagrama de classe.

```
<nodeTypes>
      <type>Class</type>
      <variant>Class</variant>
      <label alignment="center" defaultValue="Class"
         disabled="false" firstLetterCase="lowercase"
         letterCase="lowercase" pattern="(\w+)"
            position="center" marginHeight="0" marginWidth=
   "0" underlined="0"  textColor="black" textFont="11px
   Arial" >
        <field name="name" value="%1"/>
      </label>

      <containers>
           <container alignment="left" name="Attribute"
              disabled="false" pattern="(.)(\\w+):(\\w+)"
              underlined="0"
                defaultText="+attribute:type"
    numberDefaultTextbox="1" addTextBoxes="true" lineDash=
```

```
    "false">
12              <field name="visibly" value="1"/>
13              <field name="type" value="3"/>
14            </container>
15
16
17            <container alignment="left" name="Operation"
                disabled="false" pattern="(.)(\\w+)\\(\\):(\\
                w+)" underlined="0"
18                  defaultText="+operation():return"
    numberDefaultTextbox="1" addTextBoxes="true" lineDash=
    "false">
19              <field name="visibly" value="1"/>
20              <field name="type" value="3"/>
21            </container>
22
23          </containers>
24        <isConfigurable>false</isConfigurable>
25        <includeElement>false</includeElement>
26         <rotation>false</rotation>
27        <infoUrlNode>
28          <url>http://www.dcc.fc.up.pt/~zp/aulas/1617/asw/
              indice?./aulas/uml/uso/componentes/actores.html<
              /url>
29          <url>https://www.tutorialspoint.com/uml/
              uml_basic_notations.htm</url>
30        </infoUrlNode>
31        <handlers>
32           <handler  position="northwest"/>
33           <handler  position="northeast"/>
34           <handler  position="southeast"/>
35           <handler  position="southwest"/>
36        </handlers>
37        <anchors>
38          <anchor  position="north"/>
39          <anchor  position="south"/>
40          <anchor  position="east"/>
```

```
41          <anchor  position="west"/>
42        </anchors>
43        <propertiesNode>
44           <property name="type" type="String" typeShow="
               input" disabled="true" impExp="true" view="true
               "  />
45          <property name="id" type="Number" typeShow="input"
              disabled="true" impExp="true" view="true" />
46           <property name="x" type="Number" typeShow="input"
                 disabled="false" impExp="true" view="true" />
47          <property name="y" type="Number" typeShow="input"
              disabled="false" impExp="true" view="true" />
48          <property name="width" type="Number" typeShow="
              input" disabled="false" impExp="true" view="true
              " />
49          <property name="height" type="Number" typeShow="
              input" disabled="false" impExp="true" view="true
              " />
50          <property name="label" type="String" typeShow="
              input" disabled="false" impExp="true" view="true
              "/>
51        </propertiesNode>
52         <featuresNode>
53          </featuresNode>
54        <nodeStyle autoresize="true" height="40"
             iconTollbarSVGPath="./image/Class.svg" imgSVGPath=
             "./image/Class.svg"  width="80" />
55        <nodeValidation allAnchorsConnect="false" degreeIn="
             unbounded" degreeOut="unbounded">
56          <connects>
57           <connect max="unbounded" min="0" to="Class" with="
               all"/>
58           <connect max="unbounded" min="0" to="Package" with
               ="all"/>
59          </connects>
60        </nodeValidation>
61      </nodeTypes>
```

C.2 Exemplo configuração de edgeType – Extend

A seguir é apresentado a configuração `edgeType` para a aresta `Extend` do diagrama de caso de uso.

```
1
2  <edgeTypes>
3        <type>Extend</type>
4        <variant>Extend</variant>
5        <label alignment="center" defaultValue="text"
            disabled="false" firstLetterCase="" letterCase=""
            pattern=""
6             position="center" marginHeight="0" marginWidth="
     0" underlined="0"  textColor="black" textFont="11px
     Arial">
7          <field name="name" value="%1"/>
8        </label>
9        <isConfigurable>true</isConfigurable>
10       <infoUrlEdge>
11         <url> http://www.dcc.fc.up.pt/~zp/aulas/1617/asw/
              indice?./aulas/uml/uso/componentes/
              relacionamentos.html </url>
12       </infoUrlEdge>
13       <propertiesEdge>
14          <property name="type" type="String" typeShow="
              input" disabled="true" impExp="true" view="true
              "/>
15         <property name="id" type="Number" typeShow="input"
              disabled="true" impExp="true" view="true" />
16         <property name="sourceType" type="String" typeShow=
              "input" disabled="true" impExp="true" view="true
              "/>
17         <property name="targetType" type="String" typeShow=
              "input" disabled="true" impExp="true" view="true
              "/>
18         <property name="label" type="String" typeShow="
              input" disabled="false" impExp="true" view="true
              "/>
```

```
19        </propertiesEdge>
20          <featuresEdge> </featuresEdge>
21        <cardinality>false</cardinality>
22        <edgeStyle iconTollbarSVGPath="./image/UseCase/Extend
             .svg" lineDuple="false" lineDash="true">
23          <headSource dash="false" fill="false" type="none"/
               >
24          <headTarget dash="false" fill="false" type="arrow-
               open"/>
25          <stereotype name="extend" position="top"
               marginHeight="2" marginWidth="0"></stereotype>
26        </edgeStyle>
27      </edgeTypes>
```

Apêndice D

API do editor de grafo Eshu

D.1 Pacote Graph

O pacote `graph` contém as classes responsáveis pelo armazenamento, criação, alteração e remoção de nós e arestas que constituem o diagrama. Neste pacote encontram-se as classes `Graph`, `Quadtree`, `Vertice`, `ComplexVertice`, `Edge`, `TextBox` e `RectangleSelection`.

D.1.1 Classe Vertice

Classe `Vertice` é a estrutura utilizada para representar nós simples (nó com apenas uma *label* e uma imagem de fundo) no Eshu. Na tabela D.1 são apresentados os atributos da classe e na tabela D.2 as funções da classe.

Tabela D.1: API da classe Vertice – Atributos

Atributo	Descrição
x	Posição da coordenada x do nó
y	Posição da coordenada y do nó
width	Largura corrente do nó
height	Altura corrente do nó
widthDefault	Largura padrão do nó (definido no DL2)
heightDefault	Altura padrão do nó (definido no DL2)
nodeType	Objeto nodeType que guarda as informações gerais do nó
select	Seleciona o nó

anchors	Array que guarda os pontos no nó onde é possível ligar aresta e conectar com outros nós
handlers	Array que guarda os pontos no nó onde é possível selecionar e redimensionar o nó
handlerSelected	Guarda o handler selecionado
anchorSelected	Guarda o anchor selecionado
borderColor	Define a cor da borda do nó
borderWidth	Define o tamanho da linha da borda
backgroundColor	Cor de fundo do nó quando selecionado
modify	Guarda informação enviado pelo avaliador (insert, remove, modify)
degreeOut	Número do grau de saída do nó
degreeIn	Número do grau de entrada do nó
edgesConnected	Lista de arestas ligadas ao nó
label	Label do nó
rotation	Define se é possível rodar o nó ou não
angleRotation	Ângulo da rotação do nó

Tabela D.2: API da classe Vertice – Funções

Função	Discrição
Vertices	@param {name: x type: Number} @param { name: x type: Number} @param {name:id type:Number} Método construtor da classe Vertice @return null
setNodeType	@param {name: nodetype, type:Object} @return {null} Adiciona o objeto nodetype ao nó e configura o nó de acordo com as configurações do Nodetype
contains	@param {name: mx, type:Number} @param {name: my, type:Number} @return {Boolean} Verifica se o nó contém um determinado ponto

containsNode	@param {name: mx, type:Number} @param {name: my, type:Number} @return {Boolean} Verifica se um nó encontra-se dentro de outro nó
insideHandler	@param {name: mx, type:Number} @param {name: my, type:Number} @return {Boolean} Verifica se um ponto encontra-se dentro de um dos hanlders
updatePosition	@param {name: mx, type:Number} @param {name: my, type:Number} @return {Boolean} Verifica se um ponto encontra-se dentro de um dos hanlders
setX	@param {name: x, type:Number} @return {null} Define a posição x do nó
getX	@return {Number} Obtém a posição x do nó
setY	@param {name: x, type:Number} @return null Define a posição y do nó
getY	@return {Number} Obtém a posição y do nó
setWidth	@param {name: width, type:Number} @return {null} Define a largura do nó
getWidth	@return Number Obtém a largura do nó
setHeight	@param {name: width, type:Number} @return {null} Define a altura do nó
getHeight	@return {Number} Obtém a altura do nó
updateSize	@param {name: width, type:Number} @param {name: height, type:Number} @return {null} Atualiza a altura e largura do nó

selected	@return {null} Marca o nó como selecionado
unSelected	@return null Deseleciona o nó
createHandlers	@param {name: handlers, type:Json} @return {null} Define os handlers de acordo com as definidas no DL2
updateHandlres	@return {null} Atualiza a posição das handlers em relação ao nó
createAnchor	@param {name: anchors, type:Json} @return {null} Define as anchors de acordo com as definidas no DL2
updateAnchor	@return {null} Atualiza a posição das anchors em relação ao nó
showAnchor	@param {name: ctx, type:Context 2D} @return {null} Mostra os anchor definidas no nó
insideAnchor	@param {name: mx, type:Number} @ param {name: my, type:Number} @return {Boolean} Verifica se um ponto encontra-se dentro de um dos anchors do nó
connectedTo	@param {name: nodetype, type:NodeType} @param {name: edgetype, type:EdgeType} @return {Boolean} Verifica se o nó pode conectar com outro nó
connectWith	@param {name: edgetype, type:EdgeType} @return {Boolean} Verifica se a aresta pode ligar ao nó
anchorsDistanceMin	@param {name: edgetype, type:EdgeType} @return {Boolean} Verifica se a aresta pode ligar ao nó
clone	@return Node Faz um clone do nó
getEdgesConnected	@return {Array*Edge*} Obtém a lista das arestas ligadas ao nó
setEdgesConnected	@param {name: edge, Edge} @return {null} Adiciona a aresta à lista das arestas conectadas ao nó

removeEdgesConnected	@param {name: edge, Edge} @return {null} Remove uma aresta da lista das arestas conectadas ao nó
draw	@param {name: ctx, type:Context 2D} @return {null} Desenha o nó
drawIcon	@param {name: ctx, type:Context 2D} @return {null} Desenha o ícone do nó
drawStereotype	@param {name: ctx, type:Context 2D} @return {null} Desenha o stereotype do nó, caso exista
drawBorder	@param {name: ctx, type:Context 2D} @return {null} Desenha a borda do nó
setAngleRotation	@param {name: angle, type:Number} @return {null} Define o ângulo da rotação para o nó
updateLatelPositionGlobal	@param {name: label, type:TextBox} @return {null} Atualiza a posição da label de acordo com a posição do nó e o tipo da posição definida na Dl2
adjustLabel	@param {name: angle, type:Number} @return {null} Define o ângulo da rotação
createFeatures	@param {name: features, type:JSON} @return {null} Cria as features definidas na DL2
getJson	@return {Json com as propriedades dos nós} Obtém o Json do nó (utilizado para enviar informações do diagrama ao avaliador)
getProperties	@param {name: property, type:String} @return {Valor da propriedade} Obtém o valor de propriedade do nó

isConfigurable	@return {Boolean} Verifica se o nó é configurável ou não, no caso de não ser configurável é porque o nó é um ComplexVertice.
resize	@param {name: mx, type:Number} @param {name: my, type:Number} @return {null} Redimensiona o nó de acordo com o handler selecionado.
isAutoresize	@return {Boolean} Verifica se o redimensionamento é automático
isNode	@return {true} Verifica se é um nó
isEdge	@return {false} Verifica se é uma aresta
increaseOrder	@return {null} Aumenta o nível de ordem do nó
decreaseOrder	@return {null} Diminui o nível de ordem do nó
islabelEditable	@return {Boolean} Verifica se a label é editável
isAnchorFixed	@return {Boolean} Verifica se a ligação do anchor ao nó é fixo ou se pode atualizar automaticamente
setName	@param {name: text, type:Strinig} @return {null} Redefine o nome na label principal do nó
getName	@return {String} Obtém o nome da label principal do nó

D.1.2 Classe ComplexVertice

A classe *ComplexVertice*, estende o tipo `Vertice` e contem mais atributos e funções para definir nós mais complexo. Esta classe contém uma lista de `Container` B.8. Cada `Container` é um contentor que pode ter várias *labels*. Ao contrário do `Vertice`, no `ComplexVertice` não é utilizado a imagem SVG do tipo de nó, mas sim, é desenhado um retângulo (pode ter cantos arredondados), e adicionada uma linha que separa

cada`Container`.

Tabela D.3: API da classe ComplexVertice

Função	Discrição
ComplexVertice	@param {name: x, type:Number} @param {name: y, type:Number} @param {name: id, type:Number} @return {null} Método construtor da classe ComplexVertice que estende a classe Vertice
setNodeType	@param {name: nodetype, type:Object} @return {null} Define o objeto nodetype ao nó e configura-o de acordo com as configurações do Nodetype
addElement	@param {name: x, type:Number} @param {name: x, type:Number} @return {null} Adiciona um textBox ao container se as coordenadas do parâmetro se encontrarem dentro do container
selectTextBox	@param {name: label, type:TextBox} @return {null} Seleciona um determinado textBox
getPropertyContainer	@param {name: container, type:ContainerTextBox} @param {name: index, type:Number} @return {JSON} Obtém uma estrutura JSON com as informações sobre o Container (nome, id) dos TextBox que a constitui.
getPropertyContainers	@return {JSON} Obtém uma estrutura JSON com as informações sobre todos os containers
updatePositionContainers	@return {null} Atualiza a posição do container dentro do nó

insideListTextBox	@param {name: x, type:Number} @param {name: y, type:Number} @return {Boolean} Verifica se um ponto encontra-se dentro de algum Container e TextBox
isLabelSectedEditable	@return {Boolean} Verifica se o TextBox selecionado é editável
deleteBoxContainer	@return {Null} Remove o textBox que se encontra selecionado no nó
addTextBoxContainer	@param {name: idContainer, type:String} @return {Boolean} Adiciona um determinado Textbox do nó
removeTextBoxContainer	@param {name: idContainer, type:String} @param {name: idTextBox, type:Number} @return {Boolean} Remove um determinado Textbox do nó
adjustLabel	Ajusta o tamanho do nó de acordo com o tamanho da label @return {Null}
resetContainer	@return {Null} Elimina todos os TextBox do Container
getName	@return {String} Obtém o texto da label Name do nó
setName	@param {name: text, type:String} @return {String} Insere o texto da label ‘Name’ do nó
setTextContainer	@param {name: text, type:Number} @param {name: idContainer, type:String} @param {name: idTextBox, type:Number} @return {String} Define o texto da label Name do nó

D.1.3 Classe Container

Classe `Container` é uma estrutura para representar um "container"num nó. Um exemplo de da utilização deste tipo é no `ComplexVertice` para definir o nó classe no diagrama UML de classes, em que o nome da classe é editado na *label name* do nó. O elemento atributo é um `Container` onde podemos adicionar e remover *labels* para editar elementos atributo. Na tabela seguinte é apresentados as funções da classe `ComplexVertice`.

Tabela D.4: API da classe Container

Função	Discrição
x	Posição da coordenada x do container
y	Posição da coordenada y do container
width	Largura do container
height	Altura do container
name	Nome do container que também é utilizado como identificador
defaultText	Valor default para o textBox a ser inserido no container
listTextBox	Lista dos textBoxes do Container
addTextBoxes	Permite definir se é permitido adicionar um TextBox ao Container
disabled	Permite definir se é possível editar o Container
fillstyle	Define a cor para o texto da Container
selected	Guarda o estado da seleção do Container (selecionado ou não)
ContainerTextBox	@param {name: x, type:Number} @param {name: y, type:Number} @param {name: name, type:String} @param {name: text, type:String} @return {null} Método construtor do ContainerTextBox
setConfig	@param {name: config, type:Number} @return {null} Recebe como parâmetro um Json com as configurações e configura ContainerTextBox

addTextBox	@param {id: y, type:Number} @param {name: text, type:String} @return {null} Adiciona um textBox ao Container.
removeTextBox	@param {id: index, type:Number} @return {null} Remove um textBox do Container.
getTextBox	@param {id: index, type:Number} @return {TextBox} Obtém um textBox ao Container.
setTextBox	@param {name:id , type:Number} @param {name: text, type:String} @return {null} Adiciona um texto a um determinado textbox.
contains	@param {name:x , type:Number} @param {name: y, type:Number} @return {Boolean} Verifica se um ponto se encontra dentro do Container.
draw	@param {name: ctx, type:Context2d} @return {null} Desenha o Container
updatePosition	@param {name:x , type:Number} @param {name: y, type:Number} @return {Null} Atualiza a posição do Container
insideListTextBox	@param {name:x , type:Number} @param {name: y, type:Number} @return {TextBox} Verifica se um ponto (x,y) se encontra dentro de um dos textBox do Container e, caso se encontre, retorna o textBox
getWidth	@return {Number} Retorna o comprimento máximo dos textBox
getHeight	@return {Number} Retorna a altura do textBox
reset	@return {null} Volta a criar um novo Container

isEmpty	@return {Boolean} Verifica se o Container é vazio

D.1.4 Classe TextBox

A classe `TextBox` é uma estrutura que permite criar e configurar uma *label* para os elementos nós e arestas. Além disso, é possível adicionar uma expressão regular para validar o texto da *label*. A tabela seguinte apresenta os atributos e funções desta classe.

Tabela D.5: API da classe TextBox

Função	Discrição
getCursorPosition	@return {name: value, type:Number} Retorna a posição corrente da posição do cursor
setCursorPosition	@param name: value, type:Number @return null Define a posição corrente da posição do cursor
deleteLastCharName	@return null Remove o caractere na posição corrente da posição do cursor
restartCursorPosition	@return null Atualiza a posição do cursor para a posição final do texto
selected	@return null Seleciona a label caso a variável disable não seja true
unSelected	@return null Deseleciona a label caso a variável disable não seja true
increaseColumn	Incrementa um valor à variável cursorPosition, no caso do cursorPositon se encontrar na posição final, mantém a posição
decreaseColumn	@return null Decrementa um valor à variável cursorPosition, no caso do cursorPositon se encontrar na posição inicial, mantém a posição
@return null increaseLine	incrementa o valor do número da linha na cursorCposition

@return null decreaseLine	Decrementa o valor do número da linha na cursorCposition
getCoordPosition	@return Object -line:line, column:column Retorna a linha e a coluna da posição do cursor no texto
addCharLabel	@param char @return null Insere um caractere à label
labelWithBar	@param Name: ctx, type:Context @return null Desenha a barra na posição corrente do cursor
getLabelCurrentPosition	@return String Obtém uma string da posição inicial até à posição corrente do cursor
toUpperCaseFirstLetter	@return null Converte a primeira letra da label para maiúscula
toLowerCaseFirstLetter	@return null Converte a primeira letra da label para minúscula
ToUpperCase	@return null Converte a label para maiúscula
ToLowerCase	@return null Converte a label para minúscula
contains	@param Name: mx, type:Number @param Name: mx, type:Number @return Boolean Verifica se uma posição (x,y) se encontra dentro da label
containsText	@param Name: mx, type:Number @param Name: my, type:Number @return Boolean Verifica se uma posição (x,y) se encontra dentro sobre o texto da label
drawTextBG	@param Name: ctx, type:Context @return null Desenha um retângulo com fundo azul e alfa (transparência) 0.3 à volta do texto da label que assinala-a como selecionado

drawBorder	@param Name: ctx, type:Context @return null Desenha uma borda à volta do texto da label que assinala-a como selecionado
draw	@param Name: ctx, type:Context @return null Desenha a label
getMaxWidthLine	@param Name: ctx, type:Context @return Number Obtém o comprimento máximo da measureText das linhas
getHeight	@return Number Obtém a altura da label
updatePosition	@param Name: x, type:Number @param Name: y, type:Number @return Number Atualiza a posição da label
getText	@return String Obtém o texto da label
setText	@param Name: text, type:String @return Null Define o texto da label e atualiza a posição da variável cursorPosition para o comprimento do texto
isDisable	@return Boolean Verifica se a label é disable ou não
getId	@return Number Obtém o id da label

D.1.5 Classe Graph

A classe `Graph` representa um grafo constituído por um conjunto de nós e arestas, e por funções que permitem operar sobre eles. Estas funções permitem inserir, modificar, selecionar ou remover um nó ou uma aresta do grafo, e também, exportar o grafo

para uma estrutura JSON ou importar da estrutura *JSON* para grafo. Existem ainda funções para copiar, cortar e colar um nó ou um conjunto de nós. Os nós e arestas são guardados numa estrutura do tipo texttttQuadree. A tabela seguinte apresenta as funções e atributos da classe `Graph`.

Tabela D.6: API da classe Graph

Função	Discrição
width	Largura da área do grafo
height	Altura da área do grafo
nodes	Estrutura QuadTree que guarda os nós do grafo
edges	Estrutura QuadTree que guarda as arestas do grafo
rectangleSelection	Estrutura que guarda e opera sobre um conjunto de nós selecionados
selection	Objeto que guarda um elemento selecionado no grafo, pode ser tanto um nó, uma aresta ou rectangleSelection
reducibles	Array que guarda os tipos dos nós que são reduzíveis a propriedades na avaliação do grafo
Graph	@param {Name: width, type:Number} @param {Name: height, type:Number} @return {null} Método construtor da classe grafo. Inicializa a estrutura Quadtree nodes e edges
createNode	@param {Name: type, type:NodeType} @param {Name: x, type:Number} @param {Name: y, type:Number} @param {Name: id, type:Number} @return {Vertice or ComplexVetice} Cria um nó no grafo
createNodeImport	@param {Name: nodeParam, type:JSON} @return {null} Cria um nó a partir de uma estrutura JSON
ImportDiffNode	@param {Name: nodeParam, type:JSON} @return {null} Modifica o nó com base na propriedade modify do json do nó

createEdge	@param {Name: type, type:NodeType} @param {Name: id, type:Number} @param {Name: source, type:Node} @param {Name: target, type:Node} @param {Name: points, type:Array(Box)} @return {Edge} Cria uma aresta no grafo
createEdgeImport	@param {Name: type, type:NodeType} @param {Name: edge, type:JSON} @return {null} Cria um nó a partir de uma estrutura JSON
ImportDiffEdge	@param {Name: edgeParam, type:JSON} @return {null} Edita a aresta com base no campo modify da aresta
addNode	@param {Name: node, type:Node} @return {null} Adiciona um nó no grafo
addEdge	@param {Name: edge, type:Edge} @return {null} Adiciona uma aresta ao QuadTree edges
select	@param {Name: element, type:Node/Edge} @return {null} Seleciona um elemento no grafo, pode ser tanto um nó como uma aresta
selectNode	@param {Name: node, type:Node} @return {null} Seleciona um nó no grafo
selectEdge	@param {Name: element, type:Node/Edge} @return {null} Seleciona uma aresta no grafo
unselect	@param {Name: element, type:Node/Edge} @return {null} Deseleciona um elemento no grafo, pode ser tanto um nó como uma aresta
unselectNodes RectangleSelection	@return {null} Deseleciona os nós selecionados pelo RectangleSelection

insideNode	@param {Name: x, type:Number} @param {Name: y, type:Number} @return {Boolean} Verifica se as coordenadas de um ponto (x,y) se encontram dentro de algum dos nós do grafo
insideEdge	@param {Name: x, type:Number} @param {Name: y, type:Number} @return {Boolean} Verifica se as coordenada de um ponto (x,y) se encontram dentro de algum dos nós do grafo
getSelection	@return {Node,Edge,RectangleSelection} Obtém o elemento selecionado, caso exista
deleteNode	@param {Name: node, type:Vertice/ComplexVertice} @return {Null} Remove um nó do grafo e as arestas que ligam a esse nó
deleteEdge	@param {Name: node, type:Edge} @return {Null} Remove uma aresta do grafo
deleteRectangleSelection	@return {Null} Remove os nós e arestas selecionados pelo RectangleSelection
deleteElment	@return {Null} Remove um nó ou uma aresta que se encontra selecionado ou ainda o conjunto de nós e arestas selecionadas pelo RectangleSelection
isElementSelection	@return {Boolean} Verifica se existe algum elemento (nó, aresta ou RectangleSelection) selecionado no grafo
isNodeSelection	@return {Boolean} Verifica se existe algum elemento selecionado e se é do tipo nó
isEdgeSelection	@return {Boolean} Verifica se existe algum elemento selecionado e se é do tipo aresta

isNodeSelection	@return {Boolean} Verifica se existe algum elemento selecionado e se é do tipo nó
isEdgeSelection	@return {Boolean} Verifica se existe algum elemento selecionado e se é do tipo aresta
isNodeEdgeSelection	@return {Boolean} Verifica se existe algum elemento selecionado e se é do tipo nó ou aresta
isSelectionRectangleSelection	@return {Boolean} Verifica se existe algum elemento selecionado e se é do tipo RectangleSelection
clearGraph	@return {Null} Elimina todos os nós e arestas do grafo @return Null
resizeGraph	@param {Name: width, type:Number} @param {Name: height, type:Number} @return {Null} Redimensiona o tamanho do grafo e consequentemente da `quadtree` que guarda os nós e arestas, mantendo os nós e arestas.
getAllNodes	b
getAllEdges	Obtém todas as arestas do grafo num Array @return {Array(Edges)}
copy	@return {Null} Marca um nó ou conjunto de nós (RectangleSelection) a ser copiado no grafo
paste	@return {Null} Adiciona um nó ou um conjunto de nós que tenham sido copiados do nó no grafo
getPropertyElement	@param {Name: property, type:String} @return {string,number} Retorna o valor de uma propriedade do nó ou aresta definida no DL2 como visível

setPropertyElement	@param {Name: property, type:String} @param {Name: value, type:Object} @param {Name: type, type:String} @return {null} Adiciona e atualiza o valor das propriedades do nó ou aresta definida no DL2 como visível
getGraph	@return {JSON} Obtém a estrutura JSON do grafo
setGraph	@param {Name: graph, type:JSON} @return {Null} Cria um novo grafo a partir da estrutura JSON
alignHorizontally	@return {Null} Alinha os nós selecionados pelo RectangleSelection horizontalmente
alignVertically	@return {Null} Alinha os nós selecionados pelo RectangleSelection verticalmente
getWidthElement	@return {Number} Verifica se existe elemento selecionado no grafo e se é do tipo nó, caso exista retorna o valor da largura do nó
setWidthElement	@param {Name: width, type:Number} @return {Null} Define o valor da largura para o nó selecionado no grafo
setHeightElement	@param {Name: height, type:Number} @return {Null} Define o valor da altura para o nó selecionado no grafo
getHeightElement	@return {Number} Verifica se existe elemento selecionado no grafo e se é do tipo nó, caso exista retorna o valor da altura do nó
getXElement	@return {Number} Verifica se existe elemento selecionado no grafo e se é do tipo nó, caso exista retorna o valor da posição x do nó
setXElement	@param {Name: x, type:Number} @return {Null} Define o valor da posição X para o nó selecionado no grafo

getYElement	@return Number Verifica se existe elemento selecionado no grafo e se é do tipo nó, caso exista retorna o valor da posição Y do nó
setYElement	@param {Name: y, type:Number} @return {Null} Define o valor da posição Y para o nó selecionado no grafo
getName	@return {String} Obtém o valor da label name do nó selecionado no grafo
setName	@return {Null} Define o valor da label 'name' para o nó que se encontra selecionado no grafo.
getTypeSeleted	@return {Null} Obtém o tipo do elemento que se encontra selecionado no grafo
getCardinalitySource	@return {String} Obtém o valor da cardinalidade da origem da aresta que se encontra selecionada no grafo
setCardinalitySource	@param Name: value, type:String/Number @return String Define o valor da cardinalidade da origem da aresta que se encontra selecionada no grafo
getCardinalityTarget	@return {String} Obtém o valor da cardinalidade do destino da aresta que se encontra selecionada no grafo
setCardinalityTarget	@param {Name: value, type:String} @return {String} Define o valor da cardinalidade do destino da aresta que se encontra selecionada no grafo
setNameContainer	@param {Name: text, type:String} @param {Name: idContainer, type:String} @param {Name: idTextBox, type:Number} @param {Name: ctx, type:Context2D} @return {null} Atualiza o texto para um determinado textBox do nó (ComplexVertice) que se encontra selecionado no grafo

removeTextBoxContainer	@param {Name: idContainer, type:String} @param {Name: idTextBox, type:Number} @param {Name: ctx, type:Context2D} @return {null} Remove um textBox ao nó (ComplexVertice) selecionado no grafo
addTextBoxContainer	@param {Name: idContainer, type:String} @return {null} Cria um textBox e adiciona ao nó (ComplexVertice) selecionado no grafo

D.1.6 Classe Edge

A classe `Edge` define a estrutura de uma aresta no Eshu, isto é, as ligações entre os nós. Nas tabelas D.7 e D.10 são apresentados respetivamente os atributos e as funções desta classe.

Tabela D.7: API da classe Edge – Atributos

Atributo	Discrição
id	Número identificador da aresta
x	Posição da coordenada x do retângulo da aresta na QuadTree
y	Posição da coordenada y do retângulo da aresta na QuadTree
width	Largura do retângulo que representa a aresta na QuadTree
height	Altura do retângulo que representa aresta na QuadTree
lineColor	A cor da linha da aresta
source	O nó de origem da aresta
target	O nó de destino da aresta
edgeType	Objecto edgeType que contém as principais configurações da aresta e que configura a aresta.
label	Label da aresta

stereotype	Stereotype para aresta
isCardinality	Tem valor verdadeiro se a aresta contém a cardinalidade e falso caso contrário
cardinalitySource	Cardinalidade ligado ao nó de origem
cardinalityTarget	Cardinalidade ligado ao nó de destino
sourceIsAnchorFixe	Guarda o valor configurado no Dl2 que define se a ligação as anchors no source é fixo ou não
targetIsAnchorFixe	Guarda o valor configurado no Dl2 que define se a ligação as anchors no target é fixo ou não.
order	O nível da ordem da aresta (sobreposição)

Tabela D.8: API da classe Edge – Funções

Função	Discrição
Edge	@param {Name: edgeType, type:EdgeType} @return {null} Método construtor da classe Edge
setEdgeType	@param {Name: edgeType, type:EdgeType} @return {null} Adiciona o tipo da aresta e configura de acordo com o EdgeType
selected	@return {null} Marca a aresta como selecionada e altera a cor da linha
unSelected	@return {null} Desmarca a aresta como selecionada e altera a cor da linha
selectLabel	@return {null} Marca a label da aresta como selecionada
unSelectedLabel	@return {null} Desmarca a label da aresta como selecionada
conectNodes	@return {null} Seleciona o handler do nó origem e nó destino a serem ligados pela aresta.

contains	@param {Name: mx, type:Number} @param {Name: my, type:Number} @return {Boolean} Verifica se um ponto se sobrepõe à aresta
insideLabel	@param {Name: mx, type:Number} @param {Name: my, type:Number} @return {Boolean} Verifica se um ponto se encontra dentro da label da aresta
insideHandler	@param {Name: mx, type:Number} @param {Name: my, type:Number} @return {Boolean} Verifica se um ponto se encontra dentro de um dos handler que define a aresta
insideHandlerSource	@param {Name: mx, type:Number} @param {Name: my, type:Number} @return {Boolean} Verifica se um ponto com as coordenadas x e y se encontra dentro de um dos handler que liga a aresta no nó origem
insideHandlerTarget	@param {Name: mx, type:Number} @param {Name: my, type:Number} @return {Boolean} Verifica se um ponto com as coordenadas x e y se encontra dentro de um dos handler que liga a aresta no nó destino
updateBorderSize	@return {Null} Atualiza o tamanho do retângulo que define a aresta na Quadtree de acordo com a posição final e inicial do nó
getFeature	@param {Name:name, type:String} @return {String/Number} Obtém o valor da feature da aresta
setFeatures	@param {Name:name, type:String} @return {null} Recebe a configuração Json de uma feature, cria e adiciona a aresta

draw	@param {Name:ctx, type:Context2D} @return {null} Atualiza as posições do rectângulo e desenha a aresta
drawIcon	@param {Name:ctx, type:Context2D} @return {null} Desenha o ícone da aresta
getJson	@return {JSON} Obtém a estrutura JSON da aresta
increaseOrder	@return {null} Aumenta o valor da ordem (nível da sobreposição) da aresta
decreaseOrder	Diminui o valor da ordem (nível da sobreposição) da aresta @return {null}
islabelEditable	@return {Boolean} Verifica se a label da aresta é editável
getName	@return {String} Obtém o valor do texto da label na aresta
setName	@param {Name:name, type:String} @return {Null} Define um valor para o texto da label na aresta
getType	@return {String} Obtém o tipo da aresta
getCardinalitySource	@return {String} Obtém o valor da cardinalidade ligado ao nó origem da aresta
setCardinalitySource	@param {Name:cardinality, type:String} @return {String} Define o valor da cardinalidade ligado ao nó origem da aresta
getCardinalityTarget	@return {String} Obtém o valor da cardinalidade ligado ao nó destino da aresta

setCardinalityTarget	@param {Name:cardinality, type:String} @return {String} Define o valor da cardinalidade ligado ao nó destino da aresta

D.2 Pacote Eshu

O pacote `eshu` contém as classes responsáveis pela interface do utilizador, incluindo manipuladores para interação do utilizador, métodos para exportar e importar o diagrama no formato *JSON*, métodos para apresentar *feedback* visual no editor de diagramas, entre muitos outros.

D.2.1 Classe Eshu

A classe `eshu` é responsável pela criação do editor do diagrama e operações dos eventos do editor. Esta contém um elemento `canvas` que é a área de desenho do editor, um campo do tipo `graph` que guarda a representação garfo do diagrama, um objeto do tipo `commandStack` que permite executar os comandos *Undo* e *Redo*, um objeto do tipo `FormatPanel`, que permite criar a janela de propriedades do editor. Na tabela D.11 é apresentado os atributos da classe `Eshu` e na tabela D.10 as funções.

Tabela D.9: API da classe Eshu – Atributos

Atributo	Discrição
elementDraw	Guarda o tipo do elemento selecionado na toolbar
ctx	Context2D da canvas (área de desenho do grafo)
graphState	O estado do grafo
generateID	Gerador de id do Eshu
div	Div principal do Eshu
graph	A estrutura que representa o grafo
graphEditor	Elemento div que contém a canvas do editor de diagrama
toolbar	Elemento div que apresenta os nós e arestas a serem inseridos no diagrama

canvas	Elemento que contém a área de desenho do diagrama
formatPanel	Janela da formatação do editor e edição dos seus elementos
commandStack	Objeto que guarda as ações no grafo de modo a poder realizar undo/redo
imagesSVGObjects	HasMap que guarda as imagens do diagrama definida no DL2 em formato de objeto
elementsTypes	Arraylist que guarda os nodeTypes e edgeTypes definidas para o diagrama

Tabela D.10: API da classe Eshu – funções

Função	Discrição
Eshu	@param {name: div, type:HTMLDivElement} @param {name: width, type:Number} @param {name: height, type:Number} @return {null} Método construtor do Eshu. No caso do width ou height serem nulos, atribui o valor 400.
init	@return {null} Inicializa o Editor
draw	@param {name: ctx, type:Context2D} @return {null} Desenha os nós e arestas que constituem o grafo
drawNodes	@param {name: ctx, type:Context2D} @return {null} Desenha os nós que se encontram definidos no grafo
drawEdges	@param {name: ctx, type:Context2D} @return {null} Desenha os nós que se encontram definidos no grafo
drawSimpleLine	@param {name: ctx, type:Context2D} @return {null} Desenha a linha na definição da aresta

createNodeType	@param {name: nodetype, type:JSON} @return {null} Cria e adiciona ao elementsTypes um nodetype a partir da uma configuração definida no DL2 e passado como parâmetro e em formato Json
createEdgeType	@param {name: edgetype, type:JSON} @return {null} Cria e adiciona ao elementsTypes um edgetype a partir da uma configuração definida no DL2 e passado como parâmetro e em formato Json
createNodeTypes	@param {name: nodetypes, type:ARRAY(JSON)} @return {null} Cria e adiciona ao elementsTypes um conjunto de nodetype a partir de uma lista de configurações definida no DL2 e passado como parâmetro e em formato Json
createEdgeTypes	@param {name: edgetypes, type:ARRAY(JSON)} @return {null} Cria e adiciona ao elementsTypes um conjunto edgetype a partir da uma lista de configurações de edgetype definida no DL2
addNodeType	@param {name: nodetype, type: Nodetype} @return {null} Cria o Nodetype na toolbar e adiciona à elementsTypes
addEdgeType	@param {name: edgetype, type: Edgetype} @return {null} Cria o Edgetype na toolbar e adiciona à elementsTypes
setImageSVG	@param {name: key, type: String} @return {null} Cria e adicona o objecto image à HashMap imagesSVGObjects a partir de um SVG

setImage	@param {name: imagePath, type: String} @return null Cria e adiciona o objeto imagem a HashMap imagesSVGObjects a partir de um caminho de uma imagem

Events

Tabela D.11: API da classe Events

Função	Discrição
mousedown	@param {name: event, type: Event} @return {null} De acordo com o valor da variável graphState, pode criar um nó aresta ou graph RectagleSelection ou selecionar um elemento
mousemove	@param {name: event, type: Event} @return {null} De acordo com o valor da variável graphState, move um nó ou recatgleSelection, atualiza a posição da linha da aresta, ou redimensiona um elemento
mouseup	@param {name: event, type: Event} @return {null} Trata dos eventos do mouseUp do editor de acordo com o estado do grafo
dblclick	@param {name: event, type: Event} @return {null} Utilizado para desselecionar uma aresta no momento da definição ou adicionar um novo textBox no ComplexVertice
selectElement	@param {name: element, type: Vertice/edge/RectangleSelec} @return {null} Seleciona um elemento no Eshu
resizeElement	@return {null} Redefine um elemento no editor Eshu

deleteElement	@return {null} Elimina um elemento no editor Eshu
showHideFormatPanel	Mostra ou oculta a janela de formatação e edição @return {null}
handleSpecialKeys	@return {null} Trata dos eventos para atalhos como ctrl+C e ctrl+V
handleSpecialKeys ElementSelected	@return {null} Trata dos eventos do teclado relacionados com as letras de A-Z e números 0-9
selectNodeInside RectangleSelection	@return {null} Verifica e seleciona os nós que se encontram dentro do RectangleSelection
createElement	@return {null} Cria um nó ou uma aresta de acordo com a selecção na toolbar
editableName	@return {null} Edita a label Name do nó ou grafo
selectedALL	@return {null} Seleciona todos os elementos no editor Eshu
copy	@return {null} Copia o elemento selecionado no grafo
cut	@return {null} Corta o elemento selecionado no grafo
paste	@return {null} Adiciona o elemento à qual tinha sido copiado
undo	@return Cancela a última ação realizada
redo	@return {null} Cancela a ação do Undo

D.2.2 Classe Configuration

Esta classe conforme é apresentado na tabela D.12 define as funções e atributos para configurar o estilo do editor de diagrama Eshu.

Tabela D.12: API da classe Configuration

Função	Discrição
setWidth	@param {name: width, type:Number} @return {null} Define a largura do editor Eshu
getWidth	@return {type:Number} Obtém a largura do editor Eshu
setHeigth	@param {name: height, type:Number} @return null Define a altura do editor Eshu
getHeigth	@return {type:Number} Obtém a largura do editor Eshu
setGridVisible	@param {name: gridVisible, type:Boolean} @return {null} Define a visibilidade da grelha (grid), depende do parâmetro
getGridVisible	@return {type:Boolean} Obtém o valor do campo GRIDVISIBLE no editor Eshu
setGridLineColor	@param {name: gridLineColor, type:String} @return {null} Define a cor da linha da grelha
getGridLineColor	@return {type:string (color)} Obtém a cor definida para a linha da grelha do editor Eshu
setGridSpacing	@param {name: gridSpacing, type:Number} @return {null} Define o espaço entre as linhas da grelha
getGridSpacing	@return {type:Number} Obtém o valor do espaço entre as linhas da grelha do editor Eshu

setBackgroundVisible	@param {name: visible, type:Booleam} @return {null} Define se é aplicado uma cor de fundo na área de desenho
getBackgroundVisible	@return { type:Booleam} Obtém o valor da campo que define se é aplicado uma cor de fundo na área de desenho ou não.
setBackgroundColor	@param {name: gridLineColor, type:String} @return {null} Define a cor de fundo da área de desenho
getBackgroundColor	@return { type:String} Obtém a cor definida de fundo da área de desenho
resetBackgroundColor	@param @return { null} Redefinie a cor definida de fundo da área de desenho para cor padrão - branca
setToolbarBorderWidth	@param {name: borderWidth, type:Number} @return {null} Define o tamanho da borda do toolbar
getToolbarBorderWidth	@return {type:Number} Obtém o tamanho da borda do toolbar
getToolbarBorderWidth	@return {type:Number} Obtém a cor da borda do toolbar
setToolbarBorderColor	@param {name: borderWidth, type:Number} @return {null} Define a cor da borda do toolbar
getToolbarBackgroundColor	@return {type:string} Obtém a cor da de fundo da toolbar
increaseOrder	@return {null} Aumenta o nível do elemento selecionado (sobreposição)
decreaseOrder	@return {null} Diminui o nível do elemento selecionado
setWidthElement	@param {name: width, type:Number} @return {null} Define a largura para um nó selecionado

getWidthElement	@return {type:Number} Obtém a largura de um nó selecionado
setHeightElement	@param {name: width, type:Number} @return {null} Define a altura para um nó selecionado
getHeightElement	@return {type:Number} Obtém a altura de um nó selecionado
setXElement	@param {name: value, type:Number} @return {null} Define o valor da coordenada X para um nó selecionado
getXElement	@return {type:Number} Obtém a posição X de um nó selecionado
setYElement	@param {name: width, type:Number} @return {null} Define o valor da coordenada Y para um nó selecionado
getYElement	@return {type:Number} Obtém a posição Y de um nó selecionado
alignHorizontally	@param {name: align, type:String} @return {null} Alinha horizontalmente (topo, centro e abaixo) todos os nós que se encontram selecionados no RectangleSelection
alignVertically	@param {name: align, type:String} @return {null} Alinha verticalmente (esquerda, centro e direita) todos os nós que se encontram selecionados no RectangleSelection
equalizeMax	@return {null} Redefine todos os nós selecionados dentro RectangleSelection para o tamanho do nó mínimo
equalizeMin	@return {null} Redefine todos os nós selecionados dentro RectangleSelection para o tamanho do nó mínimo

getNameNodeSeleted	@return { type:String} Obtém o texto (nome) da label do nó selecionado
addTextBoxContainer	@param {name: idContainer, type:String} @return {null} Adiciona um textBox para um nó do tipo ComplexType
removeTextBoxContainer	@param {name: idContainer, type:String}, @param{name: idTextBox, type:Number} @return {null} Remove um textBox para um nó do tipo ComplexType
resetEshu	@return {null} Reinicia o editor de diagrama Eshu
showGrid	@param {name: event, type: Event} @return {null} Apresenta uma grelha no fundo da área de desenho do editor
setPosition	@param {name: position, type:String{vertical or horizontal}} @return {null} Define a posição da toolbar no Eshu
resizeCanvasNode	@return {null} Redefine a canvas do Eshu a partir de um nó, isto é, aumenta a área da canvas caso o nó selecionado intersecta lado direito ou inferior da canvas
resizeCanvas RectangleSelection	@return {null} Redefine a canvas do Eshu a partir do RectangleSelection , isto é, aumenta a área da canvas caso o RectangleSelection intersecta lado direito ou inferior da canvas

resize	@param {name: width, type:Number} @param {name: height, type:Number} @return {null} Redefine o Eshu passando a altura e a largura
showToolbar	@param {name: value, type:Boolean} @return {null} Apresenta ou oculta a toolbar de acordo com o valor passado no value (parâmetro)

D.2.3 Classe NodeType

A Classe `Nodetype` define a estrutura que guarda os tipos de nós configurados na linguagem de configurações para os `Nodetype`, conforme é apresentado em 6.1.1.

Tabela D.13: API da classe NodeType

Atributos	**Discrição**
type	@type {String} O nome do tipo do nó
width	@type {Number} A largura default do nó
height	@type {Number} A altura default do nó
iconTollbarSVGPath	@type {Number} O caminho SVG ou o SVG do imagem ícone do nó
imgSVGPath	@type {Number} O caminho SVG ou o SVG da imagem do nó
labelConf	@type {Object} A configuração para o label 'Name'
variant	@type {Number} O nome do grupo do nó
listAnchors	@type {List} Lista com as configurações para as anchors do nó
listHandlers	@type {List} Lista com as configurações para os handlers do nó

isConfigurable	@type {Boolean} Booleano que define se o nó é configurável ou não
infoUrl	@type {List} Array com as url info do nó
properties	@type {List} Lista das propriedades a serem apresentadas na janela de propriedades do nó
propertiesImpExp	@type {List} Lista das propriedades a serem importadas e exportadas no JSON do nó
features	@type {List} Lista das configurações das features do nó
connects	@type {List} Lista com os tipos de aresta e nó a que se podem ligar
degreeIn	@type {Number} Número máximo de grau de entrada para o nó
degreeOut	@type {Number} Número máximo de grau de saída para o nó
style	@type {Object} Configuração do estilo do nó
containers	@type {Object} Configuração para caso o tipo do nó ser Complex-Vertice
stereotype	@type {String} Configurações para stereotype do nó
includeElement	@type {Boolean} Define se um nó inclui ou não elemento, no caso de ser verdadeiro é assumido um ligação por sobreposição
autoresize	@type {Boolean} Define se é redimensionado o nó de acordo com o tamanho da label do nome
allAnchorConnected	@type {Boolean} Define se o nó tem de ter todos os anchor conectados

rotation	@type {Boolean} Define se é possível rodar o nó
anchorFixed	@type {Boolean} Define se a ligação ao Anchor é fixo

D.2.4 Classe EdgeType

A Classe `EdgeType` guarda os tipos de arestas configurados na linguagem de configurações para os `EdgeType`, conforme é apresentado em 6.1.1.

Tabela D.14: API da classe Edgetype

Atributos	Discrição
type	@type {String} O nome do tipo da aresta
iconTollbarSVGPath	@type {Number} O caminho SVG ou o SVG do imagem ícone da aresta
labelConf	@type {Object} A configuração para o label Name
lineDuple	@type {Boolean} Define se a aresta contém linha dupla ou não
lineDash	@type {Boolean} Define se a aresta a tracejado contém linha dupla ou não
variant	@type {Number} O nome do tipo da variant (grupo) da aresta
isconfigurable	@type {Boolean} Booleano que define se aresta é configurável ou não
infoUrl	@type {List} Array com as url info da aresta
properties	@type {List} Lista das propriedades a serem apresentadas na janela de propriedades da aresta

propertiesImpExp	@type {List} Lista das propriedades a serem importadas e exportadas no JSON da aresta
cardinality	@type {Boolean} Define se uma aresta contém ou não a cardinalidade
headSource	@type {Object} Configurações para o tipo da seta no origem da aresta
headTarget	@type {Object} Configurações para o tipo da seta no destino da aresta
stereotype	@type {String} Configurações para stereotype da aresta

Apêndice E

Questionário

Questionário de satisfação - Editor de diagramas

Este questionário tem como objectivo recolher informação do utilizador sobre a utilização do Enki

*Required

Feedback

Visibilidade do estado do sistema (feedback apropriado em tempo razoável)

1. **Quando solicito ajuda ao sistema a resposta é clara?** *
   Mark only one oval.
   - ◯ Nunca
   - ◯ Quase nunca
   - ◯ Regular
   - ◯ Quase sempre
   - ◯ Sempre
   - ◯ Não aplicável

2. **Que tipo de mensagens de erro preferes**
   Mark only one oval.
   - ◯ Curtas
   - ◯ Longas
   - ◯ Indiferente

3. **As mensagens de erro/ajuda sem kora são demasiado informativas? (revelam a solução)** *
   Mark only one oval.
   - ◯ Nunca
   - ◯ Quase nunca
   - ◯ Regular
   - ◯ Quase sempre
   - ◯ Sempre
   - ◯ Não aplicável

4. **As mensagens de erro/ajuda Com kora são demasiado informativas? (revelam a solução)** *
   Mark only one oval.
   - ◯ Nunca
   - ◯ Quase nunca
   - ◯ Regular
   - ◯ Quase sempre
   - ◯ Sempre
   - ◯ Não aplicável

5. **As mensagens são úteis a nível de aprendizagem? (ajudam a chegar à solução de forma autónoma)** *
   Mark only one oval.

   - Nunca
   - Quase nunca
   - Regular
   - Quase sempre
   - Sempre
   - Não aplicável

1. Visibilidade

Visibilidade do estado do sistema (feedback apropriado em tempo razoável)

6. **Quando executo uma tarefa o sistema informa sobre o que está a acontecer?** *
   Mark only one oval.

   - Nunca
   - Quase nunca
   - Regular
   - Quase sempre
   - Sempre
   - Não aplicável

7. **Os botões usados para realizar as tarefas mais importantes estão claramente identificados?** *
   Mark only one oval.

   - Nunca
   - Quase nunca
   - Regular
   - Quase sempre
   - Sempre
   - Não aplicável

8. **O estado dos botões (selecionado/não selecionado) é indicado com clareza?** *
   Mark only one oval.

   - Nunca
   - Quase nunca
   - Regular
   - Quase sempre
   - Sempre
   - Não aplicável

2. Compatibilidade

Compatibilidade entre o sistema e o mundo real (linguagem familiar ao utilizador)

9. **Quando tento executar uma tarefa encontro rapidamente o botão pretendido?** *
   Mark only one oval.
   - ◯ Nunca
   - ◯ Quase nunca
   - ◯ Regular
   - ◯ Quase sempre
   - ◯ Sempre
   - ◯ Não aplicável

10. **A ordem dos botões está numa sequência familiar?** *
    Mark only one oval.
    - ◯ Nunca
    - ◯ Quase nunca
    - ◯ Regular
    - ◯ Quase sempre
    - ◯ Sempre
    - ◯ Não aplicável

11. **Quando pressiono um botão o resultado é o esperado?** *
    Mark only one oval.
    - ◯ Nunca
    - ◯ Quase nunca
    - ◯ Regular
    - ◯ Quase sempre
    - ◯ Sempre
    - ◯ Não aplicável

12. **Os comandos agrupados num menu pertencem todos à categoria indicada pelo texto do botão desse menu?** *
    Mark only one oval.
    - ◯ Nunca
    - ◯ Quase nunca
    - ◯ Regular
    - ◯ Quase sempre
    - ◯ Sempre
    - ◯ Não aplicável

3. Liberdade

Liberdade e controlo do utilizador (deve ser possível ao utilizador desfazer ou refazer operações)

13. **Quando cometo um erro o sistema permite voltar atrás?** *
Mark only one oval.

- Nunca
- Quase nunca
- Regular
- Quase sempre
- Sempre
- Não aplicável

14. **Posso interromper uma ação e retomá-la em qualquer instante?** *
Mark only one oval.

- Nunca
- Quase nunca
- Regular
- Quase sempre
- Sempre
- Não aplicável

15. **Posso cancelar uma operação que está a decorrer?** *
Mark only one oval.

- Nunca
- Quase nunca
- Regular
- Quase sempre
- Sempre
- Não aplicável

16. **Posso eliminar qualquer alteração que está a ser feita e voltar ao estado anterior?** *
Mark only one oval.

- Nunca
- Quase nunca
- Regular
- Quase sempre
- Sempre
- Não aplicável

4. Consistência

Consistência e padrões (o sistema deve ser compreendido indubitavelmente e seguir as convenções conhecidas)

17. **A localização de botões e janelas é mantida quando mudo de ecrã?** *
Mark only one oval.

- Nunca
- Quase nunca
- Regular
- Quase sempre
- Sempre
- Não aplicável

18. **Os botões possuem sempre o mesmo significado quando mudo o ecrã?** *
Mark only one oval.

- Nunca
- Quase nunca
- Regular
- Quase sempre
- Sempre
- Não aplicável

19. **O significado do código de cores é consistente?** *
Mark only one oval.

- Nunca
- Quase nunca
- Regular
- Quase sempre
- Sempre
- Não aplicável

20. **É possível o uso do scroll em todas as janelas?** *
Mark only one oval.

- Nunca
- Quase nunca
- Regular
- Quase sempre
- Sempre
- Não aplicável

5. Prevenção

Prevenção de erro (evitá-los)

21. **O sistema avisa quando ocorrem problemas de entrada de dados, antes de eu executar a validação?** *

Mark only one oval.

- Nunca
- Quase nunca
- Regular
- Quase sempre
- Sempre
- Não aplicável

22. **Quando faço uma validação, o sistema apresenta uma mensagem de erro se o formato de dados não é o esperado?** *

Mark only one oval.

- Nunca
- Quase nunca
- Regular
- Quase sempre
- Sempre
- Não aplicável

23. **Sou avisado pelo sistema se estou prestes a cometer um erro grave?** *

Mark only one oval.

- Nunca
- Quase nunca
- Regular
- Quase sempre
- Sempre
- Não aplicável

24. **Existe uma separação clara entre os botões que possibilitam o acontecimento de erros graves e os restantes botões?** *

Mark only one oval.

- Nunca
- Quase nunca
- Regular
- Quase sempre
- Sempre
- Não aplicável

6. Ênfase

Ênfase no reconhecimento (minimizar o esforço de memória do utilizador tornando os objetos, as ações e as opções sempre presentes)

25. **As cores usadas nos textos estão conforme as convenções aceites para o seu significado?** *

Mark only one oval.

- Nunca
- Quase nunca
- Regular
- Quase sempre
- Sempre
- Não aplicável

26. **O texto contido em cada botão transmite a ideia do que é esperado?** *

Mark only one oval.

- Nunca
- Quase nunca
- Regular
- Quase sempre
- Sempre
- Não aplicável

27. **A informação contida no ecrã encontra-se disponível no local onde eu espero?** *

Mark only one oval.

- Nunca
- Quase nunca
- Regular
- Quase sempre
- Sempre
- Não aplicável

28. **Os itens encontram-se agrupados por género em zonas lógicas distintas?** *

Mark only one oval.

- Nunca
- Quase nunca
- Regular
- Quase sempre
- Sempre
- Não aplicável

7. Flexibilidade

Flexibilidade e eficiência no uso (deve ser permitido ao utilizador personalizar ou programar ações frequentes - ex.: criar atalhos)

29. **Consigo configurar o ecrã?** *
Mark only one oval.

- Nunca
- Quase nunca
- Regular
- Quase sempre
- Sempre
- Não aplicável

30. **Existem teclas de atalho para executar as funções mais usadas?** *
Mark only one oval.

- Nunca
- Quase nunca
- Regular
- Quase sempre
- Sempre
- Não aplicável

31. **Consigo desativar temporariamente algumas funções?** *
Mark only one oval.

- Nunca
- Quase nunca
- Regular
- Quase sempre
- Sempre
- Não aplicável

8. Estética

Estética e desenho minimalista (só informação indispensável)

32. **A informação contida no ecrã é somente a que preciso?** *
Mark only one oval.

- Nunca
- Quase nunca
- Regular
- Quase sempre
- Sempre
- Não aplicável

33. **A informação contida no ecrã destaca-se do fundo?** *
Mark only one oval.

- Nunca
- Quase nunca
- Regular
- Quase sempre
- Sempre
- Não aplicável

34. **Esteticamente o sistema é agradável nos fatores: cores, brilho, etc.?** *
Mark only one oval.

- Nunca
- Quase nunca
- Regular
- Quase sempre
- Sempre
- Não aplicável

9. Ajuda aos Utilizadores

Ajuda os utilizadores a reconhecer, diagnosticar e recuperar erros (as mensagens de erro devem ser claras e sugerir soluções)

35. **As mensagens de erro/ajuda são claras e adequadas?** *
Mark only one oval.

- Nunca
- Quase nunca
- Regular
- Quase sempre
- Sempre
- Não aplicável

36. **As mensagens de erro indicam o problema com precisão?** *
Mark only one oval.

- Nunca
- Quase nunca
- Regular
- Quase sempre
- Sempre
- Não aplicável

37. **As mensagens são curtas e objetivas?** *

Mark only one oval.

- Nunca
- Quase nunca
- Regular
- Quase sempre
- Sempre
- Não aplicável

10. Ajuda com documentação

Ajuda e documentação (documentação sempre disponível)

38. **Pesquiso facilmente a informação?** *

Mark only one oval.

- Nunca
- Quase nunca
- Regular
- Quase sempre
- Sempre
- Não aplicável

39. **A função de ajuda é bem visível?** *

Mark only one oval.

- Nunca
- Quase nunca
- Regular
- Quase sempre
- Sempre
- Não aplicável

40. **A informação é precisa, completa e percetível?** *

Mark only one oval.

- Nunca
- Quase nunca
- Regular
- Quase sempre
- Sempre
- Não aplicável

11. Facilidade

Facilidade na aprendizagem

41. **O sistema é intuitivo (percebo-o facilmente)?** *
Mark only one oval.

- Nunca
- Quase nunca
- Regular
- Quase sempre
- Sempre
- Não aplicável

42. **Tenho facilidade em aprender a trabalhar com o sistema?** *
Mark only one oval.

- Nunca
- Quase nunca
- Regular
- Quase sempre
- Sempre
- Não aplicável

12. Velocidade

Velocidade de resposta do sistema

43. **O tempo de resposta para as operações realizadas é suficientemente curto?** *
Mark only one oval.

- Nunca
- Quase nunca
- Regular
- Quase sempre
- Sempre
- Não aplicável

44. **O tempo de resposta na mudança de tarefas é suficientemente curto (ex: está a ver o enunciado e volta para o editor)?** *
Mark only one oval.

- Nunca
- Quase nunca
- Regular
- Quase sempre
- Sempre
- Não aplicável

13. Fiabilidade

Fiabilidade das suas funções (com que frequência encontro dificuldades quando...)

45. **Registo-me no sistema?** *

Mark only one oval.

- Nunca
- Quase nunca
- Regular
- Quase sempre
- Sempre
- Não aplicável

46. **Visualizo o enunciado dos exercícios?** *

Mark only one oval.

- Nunca
- Quase nunca
- Regular
- Quase sempre
- Sempre
- Não aplicável

47. **Submeto diagramas para avaliação?** *

Mark only one oval.

- Nunca
- Quase nunca
- Regular
- Quase sempre
- Sempre
- Não aplicável

48. **Tento compreender as mensagens de erro resultantes da submissão de programas?** *

Mark only one oval.

- Nunca
- Quase nunca
- Regular
- Quase sempre
- Sempre
- Não aplicável

49. **Acedo à lista de soluções dos problemas?** *

Mark only one oval.

- Nunca
- Quase nunca
- Regular
- Quase sempre
- Sempre
- Não aplicável

50. **Visualizo a avaliação/classificação dos exercícios?** *

Mark only one oval.

- Nunca
- Quase nunca
- Regular
- Quase sempre
- Sempre
- Não aplicável

14. Classificação

51. **Atendendo a todos os parâmetros que analisou como classificaria o editor de diagramas?** *

Mark only one oval.

- Muito Bom
- Bom
- Suficiente
- Insuficiente
- Mau

Observações

Pontos fortes e pontos fracos

52. **Pontos fortes?**

53. **Pontos fracos?**

54. **Sugestões de melhorias?**

Referências

[1] Norhayati Mohd Ali. A generic visual critic authoring tool. In *VL/HCC*, pages 260–261, 2007.

[2] Altova. UModel - UML tool for software modeling and application development. https://www.altova.com/umodel.html. Accessed: 2016-12-11.

[3] Rajeev Alur, Loris D'Antoni, Sumit Gulwani, Dileep Kini, and Mahesh Viswanathan. Automated grading of dfa constructions. In *IJCAI*, volume 13, pages 1976–1982, 2013.

[4] Federico Bergenti and Agostino Poggi. Improving uml designs using automatic design pattern detection. In *12th International Conference on Software Engineering and Knowledge Engineering (SEKE)*, pages 336–343. Citeseer, 2000.

[5] Martin C. Carlisle, Terry A. Wilson, Jeffrey W. Humphries, and Steven M. Hadfield. Raptor: A visual programming environment for teaching algorithmic problem solving. In *Proceedings of the 36th SIGCSE Technical Symposium on Computer Science Education*, SIGCSE '05, pages 176–180, New York, NY, USA, 2005. ACM.

[6] RB Cleidson, LR Hamilton, RP Cleber, M Kleder, and F David. Using critiquing systems for inconsistency detection in software engineering models. In ,. SEKE, 2003.

[7] Helder Correia, José Paulo Leal, and José Carlos Paiva. Enhancing feedback to students in automated diagram assessment. In *6th Symposium on Languages, Applications and Technologies (SLATE'17)*, OpenAccess Series in Informatics (OASIcs). Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, forthcoming.

[8] Helder Correia, José Paulo Leal, and José Carlos Paiva. Improving diagram assessment in mooshak. Communications in Computer and Information Science (CCIS). Springer, forthcoming.

[9] ArchStudio developer's. ArchStudio 5. http://isr.uci.edu/projects/archstudio/. Accessed: 2016-12-10.

[10] The Dia Developers. Dia Diagram Editor. http://dia-installer.de/shapes/UML/index.html.en. Accessed: 2016-12-11.

[11] Christopher Douce, David Livingstone, and James Orwell. Automatic test-based assessment of programming: A review. *Journal on Educational Resources in Computing (JERIC)*, 5(3):4, 2005.

[12] Anna Eckerdal, Michael Thuné, and Anders Berglund. What does it take to learn 'programming thinking'? In *Proceedings of the First International Workshop on Computing Education Research*, ICER '05, pages 135–142, New York, NY, USA, 2005. ACM.

[13] Francis Galiegue and Kris Zyp. Json schema: Core definitions and terminology. *Internet Engineering Task Force (IETF)*, 2013.

[14] Erich Gamma. *Design patterns: elements of reusable object-oriented software*. Pearson Education India, 1995.

[15] Gliffy. Make Diagramming a Team Sport. https://www.gliffy.com/. Accessed: 2016-12-11.

[16] John Grundy and John Hosking. Softarch: Tool support for integrated software architecture development. *International Journal of Software Engineering and Knowledge Engineering*, 13(02):125–151, 2003.

[17] K. Schmid H. Eichelberger, Y. Eldogan. A comprehensive analysis of uml tools their apabilities and their compliance. *Software Systems Engineering, Institut für Informatik l Universität Hildesheim*, 2, 8 2011.

[18] Lucid Software Inc. Diagramas bem feitos. https://www.lucidchart.com/pt. Accessed: 2016-12-11.

[19] Michael C Jensen. Eclipse of the public corporation. *Harvard Business Review (Sept.-Oct. 1989), revised*, 1997.

[20] Hieke Keuning, Johan Jeuring, and Bastiaan Heeren. Towards a systematic review of automated feedback generation for programming exercises. In *Proceedings of the 2016 ACM Conference on Innovation and Technology in Computer Science Education*, pages 41–46. ACM, 2016.

[21] José Paulo Leal, Helder Correia, and José Carlos Paiva. Eshu: An Extensible Web Editor for Diagrammatic Languages. In Marjan Mernik, José Paulo Leal, and Hugo Gonçalo Oliveira, editors, *5th Symposium on Languages, Applications and Technologies (SLATE'16)*, volume 51 of *OpenAccess Series in Informatics (OASIcs)*, pages 1–13, Dagstuhl, Germany, 2016. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[22] José Paulo Leal and Fernando Silva. Mooshak: A web-based multi-site programming contest system. *Software: Practice and Experience*, 33(6):567–581, 2003.

[23] Neil AM Maiden and Alistair G Sutcliffe. Requirements critiquing using domain abstractions. In *Requirements Engineering, 1994., Proceedings of the First International Conference on*, pages 184–193. IEEE, 1994.

[24] Robin Mason and Frank Rennie. *Elearning: The key concepts*. Routledge, 2006.

[25] MKLab. StartUML 2. http://staruml.io/. Accessed: 2016-12-11.

[26] Norhayati Mohd Ali, John Hosking, John Grundy, and Jun Huh. End-user oriented critic specification for domain-specific visual language tools. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, ASE '10, pages 297–300, New York, NY, USA, 2010. ACM.

[27] IDE NetBeans. Disponível em¡ http://www.netbeans.org¿. *Acesso em*, 11, 2008.

[28] Jakob Nielsen and Thomas K Landauer. A mathematical model of the finding of usability problems. In *Proceedings of the INTERACT'93 and CHI'93 conference on Human factors in computing systems*, pages 206–213. ACM, 1993.

[29] Jakob Nielsen and Thomas K. Landauer. A mathematical model of the finding of usability problems. In *Proceedings of the INTERACT '93 and CHI '93 Conference on Human Factors in Computing Systems*, CHI '93, pages 206–213, New York, NY, USA, 1993. ACM. doi:10.1145/169059.169166.

[30] Inc No Magic. No Magic. https://www.nomagic.com/products/magicdraw.html. Accessed: 2016-12-11.

[31] Nulab. Dive into Diagramming. https://cacoo.com/. Accessed: 2016-12-11.

[32] Inc Object Management Group. Unified Modeling Language™ (UML®). http://www.omg.org/spec/UML/. Accessed: 2016-12-02.

[33] José Carlos Paiva, José Paulo Leal, and Ricardo Alexandre Queirós. Enki: A pedagogical services aggregator for learning programming languages. In *Proceedings of the 2016 ACM Conference on Innovation and Technology in Computer Science Education*, pages 332–337. ACM, 2016.

[34] José Carlos Paiva, José Paulo Leal, and Ricardo Alexandre Queirós. Learning computer science languages in enki. In *Proceedings of the 2016 ACM Conference on Innovation and Technology in Computer Science Education*, ITiCSE '16, pages 254–255, New York, NY, USA, 2016. ACM.

[35] Michael W. Godfrey PhD. My Little UML (Tools) Page. http://http://plg.uwaterloo.ca/ migod/uml.html, 2007. Accessed: 2016-12-10.

[36] Jason E Robbins, David M Hilbert, and David F Redmiles. Software architecture critics in argo. In *Proceedings of the 3rd international conference on Intelligent user interfaces*, pages 141–144. ACM, 1998.

[37] James Rumbaugh, Ivar Jacobson, and Grady Booch. *Unified Modeling Language Reference Manual, The*. Pearson Higher Education, 2004.

[38] Valerie J Shute. Focus on formative feedback. *Review of educational research*, 78(1):153–189, 2008.

[39] Josep Soler, Imma Boada, Ferran Prados, Jordi Poch, and Ramon Fabregat. A web-based e-learning tool for uml class diagrams. In *Education Engineering (EDUCON), 2010 IEEE*, pages 973–979. IEEE, 2010.

[40] Rúben Sousa and José Paulo Leal. A structural approach to assess graph-based exercises. In *International Symposium on Languages, Applications and Technologies*, pages 182–193. Springer, 2015.

[41] Cleidson RB Souza, JS Ferreira, Kléder Miranda Gonçalves, and Jacques Wainer. A group critic system for object-oriented analysis and design. In *Automated Software Engineering, 2000. Proceedings ASE 2000. The Fifteenth IEEE International Conference on*, pages 313–316. IEEE, 2000.

[42] Alistair Sutcliffe, David Benyon, A Sutcliffe, and D Benyon. *Domain modelling for interactive systems design*. Springer, 1998.

[43] Tigris.org. Welcome to ArgoUML. http://argouml.tigris.org/. Accessed: 2016-12-10.

[44] umbrello.kde.org. Welcome to Umbrello - The UML Modeller. https://umbrello.kde.org/. Accessed: 2016-12-10.